UNIVERSITÄT LEIPZIG

Fakultät für Mathematik und Informatik
Institut für Informatik

Max–Planck–Institut für
neuropsychologische Forschung

Selbstadaptierendes Suchverfahren in leichtgewichtigen XML-Datenbanken semistrukturierter Daten

Diplomarbeit zum Erwerb des
akademischen Grades "Diplom-Informatiker"

eingereicht von:

Colin Hotzky
Matrikel-Nr.: 7772796

Aufgabenstellung und Betreuung:

Prof. Dr. Klaus-Peter Fähnrich (Universität Leipzig)

Dr. Helmut Hayd (Max–Planck–Institut)

Leipzig, den 30. Mai 2003

"Semistructured data is becoming ubiquitous."

– Alin Deutsch –

"XML will be the ASCII of the Web:
basic, essential, unexciting"

– Tim Bray –

Danksagung

Ganz besonders möchte ich mich bei folgenden Personen bedanken, ohne die diese Arbeit nicht möglich gewesen wäre:

- Herrn Prof. Dr. Klaus-Peter Fähnrich, der sich sofort entschieden hat, die Betreuung dieses Themas zu übernehmen
- Herrn Dr. Helmut Hayd für seine umfassende Unterstützung und Betreuung, sowie die vielen Tipps zur Programmierung und der Erstellung dieser Arbeit
- meinem Betreuer, Herrn Dr. Antonius van Hoof, vor allem für die vielen inhaltlichen Hilfestellungen
- meinem Betreuer, Herrn Maik Thränert, für Diskussionen und Hinweise
- Herrn Frank Schumacher für seine Ratschläge
- Frank Burkhardt für die Tipps und Tricks im Umgang mit Linux
- Stefan Burkhardt für die Bereitschaft sich meinen Fragen zu stellen
- Felix Botner für die Unterstützung beim Programmieren
- meinen Eltern, die mich während meines gesamten Studiums immer unterstützt haben und mir zur Seite standen

Kurzfassung

Mit der Entwicklung der eXtensible Markup Language (XML) und deren sprunghafter Verbreitung bei der Beschreibung von Dokumenten, nicht nur für die Veröffentlichung im Internet, sind neue Möglichkeiten für die Speicherung und Verarbeitung semistrukturierter Daten entstanden. Dies hat auch seinen Niederschlag in dem sprunghaften Anstieg von XML-Datenbanken gefunden.

Führende Hersteller von relationalen und objekt-relationalen Database Management Systems (DBMS) gehen den Weg, XML-Daten in ihre bestehenden Systeme zu integrieren. Weitere Hersteller bringen speziell für XML sogenannte XML-native Datenbanken auf den Markt. Die vorliegende Arbeit beschäftigt sich mit den leichtgewichtigen XML-Datenbanken. Diese bestehen lediglich aus einer Ansammlung von originären XML-Dateien.

Für die Anfrage an XML-Datenbanken zur Datenrecherche sind eine Vielzahl von Querysprachen entwickelt worden. Bei der Extrahierung von Daten konnten bisher zwei alternative Wege eingeschlagen werden: erstens, die Integration von XML-Dokumenten in ein bestehendes DBMS und die Nutzung deren Anfragetechniken und zweitens, die Anfrage an XML-Datenbanken durch Anwendung einer XML-Anfragesprache. Beide Möglichkeiten berücksichtigen nicht in ausreichendem Maße die Historie bereits gestellter Anfragen. Die Integration von XML-Dokumenten in ein bestehendes DBMS erfordert außerdem die Aufbereitung der XML-Dokumente, sowie die Verwaltung einer komplexen Managementsoftware.

In der vorliegenden Arbeit wird ein System konzipiert und implementiert, welches für leichtgewichtige XML-Datenbanken semistrukturierter Daten bereits getätigte Anfragen bei der Recherche berücksichtigt und in sofern als selbstadaptierend bezeichnet werden kann. Es wird der Nachweis geführt, dass das vor-

gestellte System ein erheblicher Performancegewinn gegenüber herkömmlichen Systemen auszeichnet.

Im ersten Teil der Arbeit wird auf semistrukrierte Daten eingegangen und XML als Möglichkeit für deren Modellierung vorgestellt. Im Anschluss wird ein Einblick in die verschiedenen, bis heute entwickelten Anfragesprachen für XML gegeben. Damit ist die Grundlage für das im zweiten Teil der Arbeit entwickelte selbstadaptierende Suchverfahren semistrukturierter Daten in leichtgewichtigen Datenbanken gegeben, das als Softwaresystem "Automatic Search in XML" (ASIX) realisiert wurde. Abschließend wird dieses System evaluiert und ein Ausblick auf mögliche Weiterentwicklungen gegeben.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Abkürzungsverzeichnis

AMD	Advanced Micro Devices (Corporate Name)
ASCII	American Standard Code for Information Interchange
API	Application Programming Interface
ASIX	Automatic Search in XML
CERN	Conseil Européen pour la Recherche Nucléaire
CGI	Common Gateway Interface
CPAN	Comprehensive Perl Archive Network
CPU	Central Processing Unit
DB	Database
DBMS	Database Management System
DBS	Database System
DCL	Data Control Language
DDL	Data Definition Language
DML	Data Manipulation Language
DOM	Document Object Model
DTD	Document Type Definition
EBNF	erweiterte Backus-Naur Form
FTP	File Transfer Protocol
GML	Generalized Markup Language
HTML	Hypertext Markup Language
IBM	International Business Machines (Corporate Name)
IR	Information Retrieval
ISO	International Organization for Standardization
ODMG	Object Database Management Group

OEM	Object Exchange Model
OQL	Object Query Language
Perl	Practical Extraction and Report Language
PI	Processing Instruction
POD	Plain Old Documentation
SAX	Simple API for XML
SEQUEL	Structured English Query Language
SGML	Standard Generalized Markup Language
SQL	Standard Query Language
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
W3C	World Wide Web Consortium
XML	eXtensible Markup Language
XML-GL	XML Graphical Language
XQL	XML Query Language
XSL	eXtensible Stylesheet Language
XSLT	eXtensible Stylesheet Language Transformation
YATL	Yet Another Tree Language

Vorwort

Selten hat sich ein Standard so schnell durchgesetzt: Von komplexen Dokumenten bis hin zum Austausch von Daten in den verschiedensten Systemen ist die Verwendung der Auszeichnungssprache eXtensible Markup Language (XML) die erste Wahl. So vielfältig wie ihr Anwendungsbereich so unterschiedlich sind die Anforderungen an die Speicherung und Verwaltung von Dokumenten bzw. Daten. Und das hängt im wesentlich vom Inhalt ab: Ist das XML Dokument eher text- oder datenorientiert? Oftmals sind XML Dokumente aber nichts anderes als Artikel, Webseiten oder Pressemeldungen: textorientierte Darstellungen, die medien-neutral aufbereitet werden sollen. Die interne Struktur ist dann in der Regel recht einfach und besteht nur aus dem Datentyp Text.

Kapitel 1

Einleitung

1.1 Leichtgewichtige XML-Datenbanken

Der stürmische Höhenflug von XML hat insbesondere den DBMS-Markt zur Anpassung an diese Entwicklung gezwungen. Die Anbieter von relationalen und objektorientierten Datenbanksystemen haben die Funktionalität zur Verarbeitung von originären XML-Dokumenten inzwischen in Ihre Systeme aufgenommen. Damit verbunden ist aber immer die Transformation von XML in die Modellwelt der ursprünglichen Systeme.

Daneben steigt die Zahl der angekündigten oder in der Realisierung befindlichen Datenbank-Management-Systeme speziell für originäre XML-Dokumente, den so genannten nativen XML-DBMS. Mit "Tamino" von der Software AG [tam03] und dem "Extensible Information Server" von Excelon [Exc03] haben sich bereits zwei Marktführer etabliert.

Inzwischen sind aber auch große Datenbestände von originären XML-Dokumenten/ -Dateien entstanden, die außerhalb von DBMS existieren, weil bei deren Speicherung und Verarbeitung auf die zusätzlichen Features (wie z.B. transaktionsorientierte Verarbeitung, Datenschutz, ...), die ein Datenbanksystem bietet, verzichtet werden kann.

Beispiel 1.1.1

Für die XML-Datenbank der Shakespeare-Werke [sha03] ist das sicher der Fall, da deren Inhalt sich gar nicht (oder nur nach neueren Erkenntnissen über die

Werke selbst) ändern wird.

Beispiel 1.1.2
In [Pei03] wird von einer XML-Datenbank für Schnurlos-Telefone bei der Fa. Siemens berichtet, die nicht in einem DBMS, sondern in Form von XML-Dateien gespeichert ist. Einige dieser Dateien sind bis zu 10.000 Tags groß.

Für den Fall, daß Betriebs- und Dateisystemfunktionen für Speicherung und Verwaltung von XML-Dokumenten ausreichen, können originäre XML-Dokumente auch dateibasiert als Menge von Files gespeichert werden. Die Menge von XML-Files kann dann als eine "leichtgewichtige XML-Datenbank" aufgefasst werden.

Definition 1.1.1 *Eine leichtgewichtige XML-Datenbank ist eine Menge von Files, die in einem Verzeichnis eines bestehenden Dateisystems gespeichert und ihrem Inhalt nach originäre XML-Dokumente sind.*

Damit lassen sich XML-Datenbanken gemäß der unterschiedlichen Speicherung der XML-Dokumente klassifizieren in:

- Native XML-Datenbanken – Speicherung von XML-Dokumenten im Original in einem nativen XML-Datenbanksystem.
- Relationale XML-Datenbanken – Speicherung von XML-Dokumenten als Tabellen in einem relationalen Datenbanksystem.
- Objektorientierte XML-Datenbanken – Speicherung von XML-Dokumenten als Objekte in einem Objektorientierten Datenbanksystem.
- Leichtgewichtige XML-Datenbanken – Speicherung von XML-Dokumenten als Files des Betriebs- bzw. Dateisystems.

1.2 Semistrukturierte Daten

Den kommerziellen Interessen verdanken wir bei der Entwicklung neuer Applikationen immer wieder die Entstehung neuer Datenformate. Internetseiten enthalten heute eine Vielzahl von Daten wie Texte, Bilder, Audio, Video, ..., die sich eher durch Unstrukturiertheit auszeichnen, oder zumindest strukturell schwer

zugänglich sind. Für solche Daten existiert kein allumfassendes Datenmodell. Während sich die klassische Verarbeitung und Speicherung von Daten an der vollständigen Kenntnis der Datenstrukturen orientiert, wachsen mit dem Internet die Zwänge zur Verarbeitung und Speicherung von flexiblen, unregelmäßigen, nur teilweise strukturierten – eben semistrukturierten – Daten.

Eigenschaften semistrukturierter Daten

Unter semistrukturierten Daten sind nun aber nicht irgendwelche "halb strukturierten" Daten zu verstehen. Unter diesem Begriff werden Daten mit ganz speziellen Eigenschaften verstanden [RV03].

1. Zunächst wird gefordert, dass die Struktur solcher Daten zwar nicht a priori bekannt sein muss, so doch wenigstens in den Daten selbst eingebettet ("embedded") sein soll. Da die Struktur aus den Daten selbst hervorgeht, liegen also bezüglich der Strukturierung selbstbeschreibende Daten vor.

2. Die Struktur der Daten darf nur teilweise bekannt sein und muss nicht notwendig zu einem im voraus definierten Schema passen, wie es bei Datenbanken mit fester relationaler oder objektorientierter Struktur der Fall ist. Die Daten müssen also keinem strengen Typsystem folgen. Ähnliche Inhalte können durch unterschiedliche Typen repräsentiert werden. Andererseits ist es aber auch durchaus zulässig, eine explizite Strukturbeschreibung anzugeben, wie später mit XML Document Type Definitions gezeigt wird.

3. Die Struktur der Daten kann **irregulär** sein. Man kann unterscheiden in:

   - **statische Irregularität** und
   - **dynamische Irregularität**.

   Es können Attribute zu einem Objekt vorhanden sein, während das bei einem anderen Objekt nicht der Fall ist oder inhaltlich gleiche Attribute können verschieden in unterschiedlichen Objekten auftreten. Auch können Daten einfach fehlen, wodurch die Struktur aufgrund der Selbstbeschreibung durch die Daten nur teilweise strukturiert ist. In diesem Fall spricht man von statischer Irregularität.

Unterliegen die Datenstrukturen auch einer zeitlichen Änderung – damit letztlich auch das beschreibende Datenschema einer so genannten Schemaevolution – wird dies als dynamische Irregularität bezeichnet.

4. Benutzt man das generische Graphmodell zur Ableitung eines Datenmodells aus der eingebetteten Datenstruktur, lässt sich diese modellieren, indem die Kanten die Struktur und die Knoten die Daten bilden. Da der Weg zu einem Datum dann als Pfad bezeichnet werden kann, spricht man beim Datenzugriff vom **pfadorientierten Zugriff**.

Bei Zusammenfassung der angegebenen Eigenschaften kann man folgende Definition für semistrukturierte Daten angeben:

Definition 1.2.1 *Daten mit eingebetteter oder selbstbeschreibender Struktur, die mittels eines graphbasierten Datenmodells modellierbar sind, heißen* ***semistrukturierte Daten****.*

1.3 Motivation und Zielsetzung

Der Drang zur Bereitstellung der verschiedensten Informationen im World Wide Web oder in einem Intranet hat unter anderem auch dazu geführt, sich Datenmodellen zuzuwenden, die eine Darstellung unterschiedlich strukturierter Daten ermöglichen. Im semantischen Umgang mit diesem neuen Typ semistrukturierter Daten hat sich die Auszeichnungssprache eXtensible Markup Language (XML) durchgesetzt. Diesem Trend folgend haben die Hersteller bekannter Datenbankmanagementsysteme (DBMS) die Verarbeitung solcher Daten inzwischen in ihre Systeme integriert. Die Integration in DBMS aber ist für viele XML-Datenbanken unrentabel. Außerdem sind DBMS nicht portabel und sehr aufwendig in der Administrierung. Lohnt sich die Anschaffung eines DBMS nicht, können die Daten auch auf sehr einfache Weise in XML-Dateien gespeichert werden. Der direkte Zugriff und die Originalstruktur bleiben erhalten.

Bei DBMS bleibt bei der Informationsgewinnung aus XMLstrukturierten Daten die Historie bereits getätigter Anfragen unberücksichtigt. Vorteile aus dem Systemcache können nur für unmittelbar nacheinander gestellte Anfragen gezogen

werden. Die Bearbeitung semistrukturierter Daten ohne Einsatz eines DBMS, also lediglich als eine Ansammlung einer Menge von Files, einer so genannten **leichtgewichtigen Datenbank**, erfreut sich in jüngster Zeit zunehmender Beliebtheit. Das Antwortzeitverhalten ist bei Informationsanfragen an eine solche Datenbank stark abhängig von der Anzahl der enthaltenen XML-Files, die auf einige tausend anwachsen kann. Es ist deshalb von Interesse, ob sich der Einsatz eines intelligenten, automatisierten Suchverfahrens für die Informationsgewinnung lohnt und diese wesentlich beschleunigen kann.

1.4 Aufbau dieser Arbeit

Nach den einführenden Bemerkungen im ersten Kapitel zu leichtgewichtigen XML-Datenbanken und semistrukturierten Daten steht das zweite Kapitel ganz im Zeichen von XML. Im dritten Kapitel wird auf verschiedene Anfragesprachen eingegangen, sowie auf alternative Ansätze für die Suche in XML-Files.

Das vierte Kapitel behandelt die Entwicklung eines selbstadaptierenden Suchverfahrens in leichtgewichtigen XML-Datendanken. Dessen Umsetzung und Implementierung im Softwaresystem "Automatic Search in XML" (ASIX) steht im Mittelpunkt des fünften Kapitels. Im sechsten Kapitel schließt sich eine Performanceanalyse an.

Schließlich gibt das siebente Kapitel eine Zusammenfassung der Arbeit. Es wird der Entwicklungsstand der Implementierung erläutert und ein Ausblick auf weiterführende Arbeiten gegeben.

Kapitel 2

XML

2.1 Das World Wide Web Consortium

W3C Im Oktober 1994 wurde ein Konsortium gegründet, welches allgemeine Protokolle für das World Wide Web entwickeln sollte. So entstand der Name *World Wide Web Consortium*, kurz W3C. Die Protokolle sollen die Interoperabilität von Anwendungen im Web sichern. Tim Berners-Lee war maßgeblich an der Gründung beteiligt. Zusammen mit dem Conseil Européen pour la Recherche Nucléaire (CERN) gründete er das W3C am Massachusetts Institute of Technology. Das W3C erreichte schnell eine große Bedeutung und hat bis heute mehr als 400 Mitgliedsorganisationen.

Empfehlungen Das W3C verabschiedet Empfehlungen, die den Charakter von Standards haben. Auf den Webseiten des Konsortiums können alle Empfehlungen und andere veröffentlichte Dokumente abgerufen werden. Die *Hypertext Markup Language* (HTML) und die *eXtensible Markup Language* (XML) gehören auch zu diesen Empfehlungen. Bis aus einem Vorschlag (*Note*) eine Empfehlung (*Recommendation*) wird, muss er noch drei weitere Stati durchlaufen. Im *Process Document* des W3Cs werden folgende Typen technischer Reporte definiert, Zitat [w3c03]:

- *Notes:* A Note is a dated, public record of an idea, comment, or document. A Note does not represent commitment by W3C to pursue work related to the Note.

- *Working Drafts:* A Working Draft represents work in progress and a commitment by W3C to pursue work in this area. A Working Draft does not imply consensus by a group or W3C.

- *Candidate Recommendations:* A Candidate Recommendation is work that has received significant review from its immediate technical community. It is an explicit call to those outside of the related Working Groups or the W3C itself for implementation and technical feedback.

- *Proposed Recommendations:* A Proposed Recommendation is work that (1) represents consensus within the group that produced it and (2) has been proposed by the Director to the Advisory Committee for review.

- *Recommendations:* A Recommendation is work that represents consensus within W3C and has the Director's stamp of approval. W3C considers that the ideas or technology specified by a Recommendation are appropriate for widespread deployment and promote W3C's mission.

Alles beginnt mit einer *Note*. Damit wird das W3C über ein bestimmtes Thema informiert. Ob das W3C aufgrund eines Vorschlags in Aktion tritt, liegt allein beim W3C selbst. Wird eine Arbeitsgruppe gebildet, wird sie sich damit befassen, ein *Working Draft* zu erstellen. Wenn alle Unklarheiten beseitigt sind und alle Verbesserungsvorschläge in den Working Draft eingebunden wurden, wird daraus eine *Candidate Recommendation*. Nun müssen Entwickler, die im Arbeitspapier spezifizierten Funktionen implementieren und ihre Erfahrungen an die Arbeitsgruppe des W3Cs weiterleiten. Bei tiefgreifenden Problemen kann es auch sein, dass das Arbeitspapier zum Working Draft zurückgestuft wird. Das Papier erreicht den Status *Proposed Recommendation*, wenn genügend positive Erfahrungen und Berichte vorliegen. Nun wird das Arbeitspapier dem "Advisory Committee" vorgelegt. Jede der am W3C beteiligten Organisationen stellt ein Mitglied, dass das Arbeitspapier in seiner Organisation promoten soll. Zwei Wochen nach dem Ablaufen der Frist für das Einreichen von Kommentaren erreicht das Projekt den Status *Recommendation*. [KM03], [w3c03]

2.2 Historie

SGML Im Jahr 1969 wurde bei IBM die Generalized Markup Language (GML) entwickelt. Es war eine Meta-Sprache, mit der Strukturen beliebiger Datenmengen beschrieben werden konnten. Später wurde aus GML SGML (S steht für Standard) und sie bekam 1986 von der International Organization for Standardization (ISO) die Nummer ISO 8879. Damit wurde SGML zu einem internationalen Standard für die Speicherung und den Austausch von Daten. Jedoch die hohen Kosten für eine Implementation eines SGML-fähigen Systems und seine Komplexität verhinderten eine große Ausbreitung. Mitte der neunziger Jahre verstärkte sich das Verlangen nach einer einfacheren Lösung, die auch internettauglich sei.

XML Im Jahr 1996 begann das World Wide Web Consortium (W3C) eine Sprache zu entwickeln, die genauso mächtig und flexibel ist wie SGML, aber auch so breit akzeptiert werden sollte, wie es damals und heute auch die Hypertext Markup Language (HTML) war und ist. Die heutige Form von XML ist die, welche 1998 vom W3C empfohlen wurde. Die große Popularität von XML ist darauf zurückzuführen, dass sie einfacher ist als SGML. Durch die weltweite Vernetzung von Rechnersystemen und durch das World Wide Web ist der Bedarf an Informationsaustausch sehr groß geworden. Austauschformate müssen leicht anwendbar und verständlich sein, aber auch von Programmen ausgewertet werden können. Durch die Ähnlichkeit zu HTML wirkt XML vielen Anwendern vertraut. Das stellt einen weiteren Vorteil dar. XML erlaubt die Definition anwendungsspezifischer Grammatiken, welches in HTML nicht möglich ist. Abbildung 2.1 auf der nächsten Seite stellt die Entwicklung von XML graphisch dar, [ABK00], [KM03].

2.3 Merkmale und Syntax von XML

2.3.1 Überblick

Entwicklungsziele Bei der Entwicklung von XML wurden einige wesentliche Ziele verfolgt, die das Ergebnis maßgeblich beeinflußten. Demnach soll XML über das Internet und auf einfache Weise nutzbar sein und eine große Bandbreite von Anwendungen unterstützen. XML soll weiterhin zu SGML kompatibel sein. Die

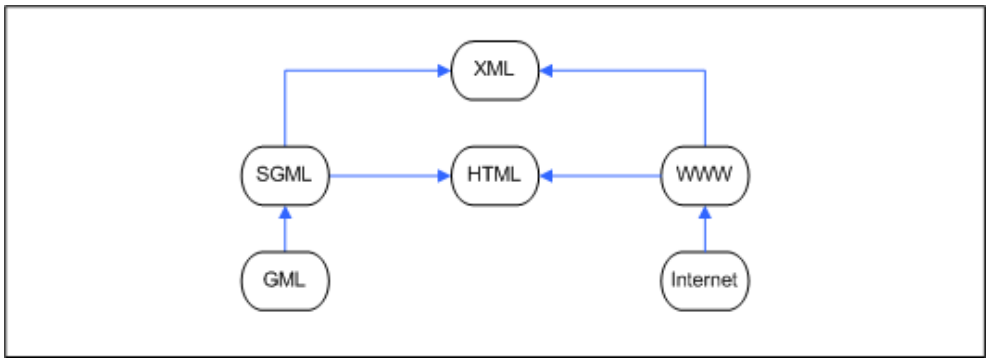


Abbildung 2.1: Die Wurzeln von XML

XML-Files und die Programme, die sie auswerten, sollten einfach zu erstellen sein. Auch die Lesbarkeit für den Menschen ist ein wichtiges Ziel.

Merkmale XML ist eine Markup-Sprache, weil Daten und Strukturinformationen in demselben Dokument abgespeichert werden. Diese Eigenschaft wird auch als *selbstbeschreibend* bezeichnet. Das ist ein Grund, warum XML für viele Anwendungen gewählt wird. Ein weiteres Merkmal von XML ist die Lesbarkeit. Menschen, wie auch Maschinen, können in einfacher Weise Informationen aus den Dokumenten erhalten. XML ermöglicht sehr gut die Darstellung und Modellierung semistrukturierter Daten. Durch Anwendung von so genannten *Style Sheets* ist XML auch als Zwischenformat zur Präsentation von Daten geeignet.

2.3.2 Syntax

Tags

XML-Dokumente haben dieselbe Struktur wie HTML-Dokumente, d.h. es gibt Tags, die mit einer öffnenden spitzen Klammer (<) beginnen und mit einer schließenden spitzen Klammer (>) enden. Die wesentlichsten Vorteile von XML gegenüber HTML sind die Trennung von Form und Inhalt, und dass Metainformationen in den Attributen untergebracht werden können. Bei den Tags muss die Groß- und Kleinschreibung beachtet werden. `<P>` und `<p>` werden somit als verschiedene Tags gewertet, denn die Schreibweise ist casesensitiv. Richtig ist `<p>...</p>`.

Tags können im Prinzip beliebige Zeichenketten sein. Es existieren zwei Ein-

schränkungen: Die Namen müssen mit einem Buchstaben, einem Unterstrich oder mit einem Doppelpunkt anfangen. Sie dürfen nicht mit *XML*, *xml* oder einer Abwandlung dieser Buchstaben (*xMl,...*) beginnen.

Um auf dem internationalen Markt zu bestehen, unterstützt XML nicht nur den 7-Bit-ASCII-Zeichensatz, sondern alle Zeichen, die im 16-Bit-Unicode-2.1-Zeichensatz enthalten sind.

Das Listing 2.1 zeigt, wie ein XML-Dokument aussehen kann:

Listing 2.1: personal.xml

```
<?xml version='1.0'?>
<!-- This file represents a fragment of a user database -->
<group specialty='IT-Manager'>
   <assistant>
      <name>
         <firstname>John</first_name>
         <lastname>Smith</last_name>
      </name>
      <private>
         <call_numbers>
            <number>00493411245632</number>
            <number>00491798545212</number>
         </call_numbers>
         <age>26</age>
      </private>
      <business>
         <company>XML-Soft</company>
         <bureau>300</bureau>
         <call_numbers>
            <number>00493411111110</number>
         </call_numbers>
      </business>
   </assistant>
</group>
```

Attribute

Tags und deren Inhalt beherbergen oft nicht genügend Informationen. Mit Hilfe von Attributen kann dem Inhalt eines Elements eine Bedeutung zugewiesen werden. Das Tag in Beispiel 2.3.1 auf der nächsten Seite gibt keine Auskunft, in welcher Währung der Preis angegeben ist. Ein Attribut `currency` kann diese Aufgabe übernehmen, wie es in Beispiel 2.3.2 auf der folgenden Seite gemacht

wurde.

Beispiel 2.3.1 `<price>20</price>`

Beispiel 2.3.2 `<price currency="EUR">20</price>`

Attribute stehen innerhalb eines Tags und müssen immer gequotet werden.

Attributtypen In einer Document Type Definition (DTD) (siehe Abschnitt 2.3.3 auf Seite 16) kann über die Attributlistendeklaration festgelegt werden, welche Attribute ein Element haben muss und welche optional sind. Die folgende Übersicht fasst das nötige Wissen zur Deklaration von Attributen zusammen:

1. Attribute müssen einen Namen haben.
2. Es existieren verschiedene Typen für Attribute, die folgendermaßen definiert sind:
   - *CDATA/PCDATA:* Dieser Datentyp kann beliebige Inhalte enthalten, da er nicht weiter ausgewertet wird.
   - *ID:* Diese Werte dienen zur Identifikation eines Dokumentteils und müssen deshalb eindeutig sein.
   - *IDREF/IDREFS:* Mit diesen Attributen kann auf eine oder mehrere IDs innerhalb eines Dokuments verwiesen werden und somit auch auf verschiedene Dokumentteile.
   - *ENTITY/ENTITIES:* Der Attributwert ist eine Referenz auf eine oder mehrere Entities.
   - *NMTOKEN/NMTOKENS:* `NMTOKENS` sind Strings, die die gleichen Bedingungen wie Element- oder Attributnamen erfüllen müssen. Sie können mit einem Doppelpunkt, einem Unterstrich oder einem Buchstaben beginnen. Anschließend dürfen sie diese Zeichen und auch Ziffern enthalten.
   - *(wert1 | wert2 | ...):* In runden Klammern, von senkrechten Strichen getrennt, kann eine Aufzählung gültiger Werte eines Attributs erfolgen. Auch hier gelten die Konventionen für Elementnamen.

3. Es werden drei Arten unterschieden, die die Angabe der Attribute bestimmen:
   - `#REQUIRED` — das Attribut muss angegeben sein
   - `#IMPLIED` — das Attribut ist optional
   - `#FIXED` — der in der DTD spezifizierte Defaultwert wird eingesetzt
4. Der Defaultwert wird immer dann verwendet, wenn ein Attribut als `#FIXED` deklariert ist.

Die Deklaration von Attributen in einer DTD wird so angegeben:

Beispiel 2.3.3

```
<!ATTLIST specialty CDATA #REQUIRED floor CDATA #IMPLIED>
```

Entities

Entity In größeren Dokumenten kann es vorkommen, dass mehrere Teile immer wiederkehren. Diese Teile können als *Entities* deklariert werden. Sobald sie im Dokument aufgerufen werden, werden sie mit Hilfe der XML-Prozessoren[1] durch den entsprechenden Inhalt ersetzt.

Typen Es wird zwischen *allgemeinen Entities* und *Parameter-Entities* unterschieden. Letztere werden ausschließlich in DTDs eingesetzt. Weiterhin wird zwischen *internen* und *externen* Entities unterschieden. Interne Entities werden in der gleichen Datei und externe in einer anderen deklariert.

Die folgenden zwei Beispiele zeigen Deklaration und Aufruf eines Entities:

Beispiel 2.3.4 `<!ENTITY Entityname 'Ersetzungstext'>`

Beispiel 2.3.5 `&Entityname;`

Anstelle von Hochkommata sind auch Anführungsstriche als Quoting des Ersetzungstextes zulässig.

[1] siehe Abschnitt 2.3.2 auf Seite 16

Vordefinierte Entities Zeichen, die in XML bereits eine besondere Funktion besitzen, können mit Entities ausgedrückt werden. Das betrifft folgende Zeichen:

- `<` = <
- `>` = >
- `&` = &
- `'` = ´
- `"` = "

Das nächste Beispiel zeigt die Anwendung dieser Entities:

Beispiel 2.3.6

```
<business>
        <company>&quot;XML-Soft&quot;</company>
              ⋮
</business>
```

Character-Entities Mit der Angabe von Dezimalzahlen oder Hexadezimalzahlen kann auf die Zeichen der erweiterten ASCII-Menge (ISO 8859/1 oder auch Latin-1 genannt) und auf die Zeichen des Unicode-Zeichensatzes (ISO 10646) zugegriffen werden.

Beispiel 2.3.7

dezimale Angabe für <: `<`

hexadezimale Angabe für <: `<`

Nichtgeparste Entities Diese Entities werden auch als binäre Entities bezeichnet. Sie kommen vor allem zum Einbinden von Bilddaten zum Einsatz, weil die Werte nicht im XML-Format sein müssen. Nichtgeparste Entities können nur in `ENTITY`- und `ENTITIES`-Attributen benutzt werden und müssen immer extern deklariert sein. Die Komponente `NOTATION` zeigt an, dass die Daten als nichtgeparstes Entity zu identifizieren sind. Außerdem kann angegeben werden, mit

welcher Applikation die Daten ausgeführt werden sollen. Die nächsten beiden Beispiele zeigen die Syntax der binären Entities und die Pflichtangabe, wenn keine Informationen zur Ausführung zur Verfügung stehen.

Beispiel 2.3.8 *Angabe zur Anzeige von JPEG-Bilddaten*

```
<!NOTATION JPEG SYSTEM 'showjpeg.exe'>
```

Beispiel 2.3.9 *Angabe, wenn keine weiteren Informationen verfügbar sind*

```
<!NOTATION TEX SYSTEM ''>
```

Parameter-Entities Auch in DTDs ist es wünschenswert, sich wiederholende Teile ersetzen zu lassen. Parameter-Entities werden dazu benutzt. Das beste Anwendungsbeispiel sind Adressdaten von Personen, die immer dieselbe Struktur haben. Die folgenden Beispiele zeigen Deklaration und Aufruf eines internen Parameter-Entity.

Beispiel 2.3.10

```
<!ENTITY % address_definition "(city,postcode,street,number)">
```

Beispiel 2.3.11

```
<!ELEMENT address %address_definition;>
```

Externe Parameter-Entities werden, wie folgt, deklariert und eingebunden:

Beispiel 2.3.12

```
<!ENTITY % testent SYSTEM "http://www.testurl.ts/testent.ent">
```

Beispiel 2.3.13

```
%testent;
```

Processing Instructions

Durch die Anwendung der *Processing Instructions* (PI) kann ein Dokument noch um Anweisungen zur Verarbeitung ergänzt werden. Im Namen einer Processing Instructions darf dabei in keiner Weise der Teilstring `"xml"` auftauchen. Die häufigste Anwendung ist die Zuweisung von Style Sheets. Beispiel 2.3.14 zeigt die Anwendung:

Beispiel 2.3.14

```
<? PI-Name PI-Anweisung ?>
<?xml-sylesheet type="test/xsl" href="/home/hotzky/xml/test.xsl"?>
```

Kommentare

Die Anwendung von Kommentaren ist vielseitig. Sie werden zur Strukturierung von XML-Dokumenten und DTDs oder für Zusatzinformationen genutzt. Kommentare können auch von Applikationen ausgewertet werden. Mit der eXtensible Stylesheet Language (XSL) können Kommentare ausgewertet und dargestellt werden. Auch mit Parsern[2], die das Document Object Model (DOM) verwenden, lassen sie sich erfragen, manipulieren und an Applikationen weiterleiten. Dennoch sollten sie nicht zur Angabe von Informationen gehören, die in das Dokument oder in die Processing Instructions gehören.

Namensräume

Es kann in bestimmten Fällen von Nutzen sein, Elemente und Attribute in Gruppen zu ordnen. Namensräume werden hierfür verwendet. Die Deklaration geschieht durch ein reserviertes Attribut in der Form `xmlns:prefix=URI`[3]. Das `prefix` wird zur Identifikation des Namensraumes innerhalb eines Dokuments benutzt. Die `URI` muss ein eindeutiger Bezeichner sein. Deshalb werden dafür in der Regel URLs[4] verwendet. Bei der Verarbeitung eines XML-Dokuments kann so zum Beispiel beim Auftreten eines Namensraumes vom Standardformat in ein

[2]Ein Parser ist ein Programm, welches ein Dokument zur Verarbeitung des aufrufenden Programms vorbereitet.

[3]URI = Uniform Resource Identifier

[4]Uniform Resource Locator

spezielles anderes Format gewechselt werden. Werden Namensräume verwendet, können DTDs nicht genutzt werden, da es in DTDs keine Namensraumdeklarationen gibt. Dies ist darauf zurückzuführen, dass DTDs aus SGML stammen und es zu dieser Zeit noch keine Namensräume gab.

XML-Prozessoren

Es existieren zwei verschiedene Arten von XML-Prozessoren, die XML-Dokumente parsen und die Informationen aus dem XML-Dokument an die Applikation weiterleiten, die den XML-Prozessor aufgerufen hat – validierende und nichtvalidierende Parser. Validierende Parser testen, ob die XML-Dokumente den Vorschriften der zugehörigen DTD genügen.

2.3.3 Document Type Definitions

Document Type Definitions, kurz DTD, sind Dokumente, die in einer Markup-Sprache geschrieben sind. Die Syntax solcher DTDs ist in der XML-Spezifikation festgelegt. In DTDs existieren nur Deklarationen, die die zulässigen Elemente, Attribute, Entities und Notationen der zugehörigen XML-Dokumente festlegen. Jede Deklaration beginnt mit der Zeichenfolge `<!`.

Beispiel 2.3.15

```
<!ELEMENT address (street,postcode,city,country)>
<!ATTLIST address location (private|business) 'business'>
          ⋮
```

2.3.4 Wohlgeformtes XML-Dokument

Ein wohlgeformtes XML-Dokument ist ein Dokument, das genau der Spezifikation und der darin festgelegten Syntax entspricht. Um die Wohlgeformtheit zu erreichen, muss ein XML-Dokument in bestimmte Teile gegliedert sein und bestimmte Eigenschaften haben.

Prolog, Rumpf und Epilog

Ein XML-Dokument ist in maximal drei Teile gegliedert. Prolog und Epilog sind optional und treten am Anfang bzw. am Ende des Dokuments auf. Der Hauptteil ist der Rumpf, der die eigentlichen Daten enthält. Prolog und Epilog enthalten Zusatzinformationen (siehe Abb. 2.2). Beim Rumpf muss die Schachtelung der Elemente strikt eingehalten werden. Ein Element, welches innerhalb eines anderen geöffnet wurde, muss als erstes wieder geschlossen werden. Die Struktur muss also einem hierarchischen Baum entsprechen (Bsp. 2.3.19 auf der nächsten Seite).

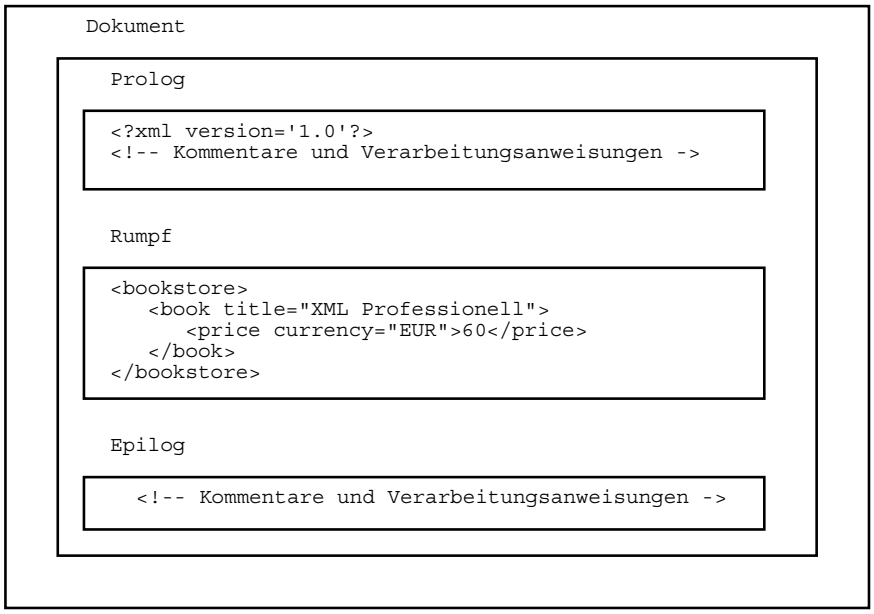


Abbildung 2.2: wohlgeformtes XML-Dokument

Prolog Der Prolog wird mit `<?` eingeleitet und enthält Informationen über die verwendete XML-Version, die Kodierung des Dokuments und ob dem Dokument eine externe Markup-Deklaration zugeordnet wird. Dies kann zum Beispiel eine Document Type Definition (DTD) sein. Listing 2.1 auf Seite 10 enthält ein vollständiges XML-Dokument, auf das auch in anderen Teilen der Arbeit Bezug genommen wird. Unter `version` wird die Version von XML angegeben. Optional ist die Angabe von `encoding`. Wurde nichts angegeben, ist `UTF-8` der Default-

wert. Andere mögliche Angaben sind beispielsweise `Latin-1` und `UTF-16`. Mit `standalone` (optional) wird spezifiziert, ob dem Dokument zum Beispiel eine DTD[5] zugeordnet ist oder nicht.

Beispiel 2.3.16

```
<?XML version="1.0" encoding="UTF-16" standalone="yes">
```

Beispiel 2.3.17

```
<?XML version="1.0" encoding="UTF-8" standalone="no">
```

Beispiel 2.3.18

```
<!DOCTYPE test-system SYSTEM "test.dtd">
```

Rumpf Der Rumpf enthält die eigentlichen Daten eines XML-Dokuments. Hier stehen die Elemente und deren Inhalt. Die Schachtelung der Elemente muss strikt eingehalten werden, d.h. ein Element, welches innerhalb eines anderen geöffnet wurde, muss wieder geschlossen werden, bevor das andere geschlossen wird. Die Struktur muss also einem hierarchischen Baum entsprechen (Bsp. 2.3.19).

Beispiel 2.3.19

richtig: `<book><title>\ldots</title></book>`

falsch: `<book><title>\ldots</book></title>`

Epilog Mit dem optionalen Epilog kann ein Dokument abschließen. Hier stehen Kommentare und Verarbeitungsanweisungen. Kommentare werden wie in HTML mit `<!-- ... -->` angegeben (Bsp. 2.3.20), Verarbeitungsanweisungen mit `<? ... ?>` (Bsp. 2.3.14 auf Seite 15).

Beispiel 2.3.20

```
<!-- Hier kann beliebiger Text stehen -->
```

[5] siehe Abschnitt 2.3.3 auf Seite 16

Eigenschaften von XML

Ein XML-Dokument muss folgende Eigenschaften besitzen, um als wohlgeformt bezeichnet werden zu können. Jedes Starttag muss auch ein Endtag haben, es sei denn, es handelt sich um ein leeres Tag (`<leerestag/>`). Dabei muss die korrekte Verschachtelung immer eingehalten werden. Die spitzen Klammern sind für die Tags reserviert. Tauchen diese Klammern an anderen Stellen im Dokument auf, müssen sie durch die entsprechenden Entities `<` und `>` ersetzt werden. Alle Attributwerte müssen in Hochkommata oder Anführungszeichen gefasst werden. Existiert keine DTD zu einem Dokument, haben alle Attribute den Wert `CDATA`. Die Namen aller Attribute innerhalb eines Elements sind eindeutig. Kein Attributwert darf eine sich öffnende spitze Klammer oder einen Verweis auf ein externes Entity enthalten. Jedes Dokument darf nur genau ein Wurzelelement besitzen, dessen Namen im gesamten Dokument eindeutig ist. Jedes geparste Entity, welches im XML-Dokument indirekt oder direkt referenziert wird, ist selbst auch wohlgeformt. Ein Dokument ist nur dann ein XML-Dokument, wenn es die genannten Anforderungen erfüllt. Desweiteren heißt ein XML-Dokument gültig, wenn alle Elemente und Attribute eines Dokuments in einer DTD in der entsprechenden Gruppierung deklariert wurden. Beide Eigenschaften, Wohlgeformtheit und Gültigkeit, werden mit XML-Prozessoren überprüft.

2.4 Application Programming Interfaces für XML

2.4.1 Simple API for XML

Die Simple API for XML (SAX) ist eine eventorientierte Schnittstelle zwischen XML-Dokumenten und Anwendungen, die auf sie zugreifen. SAX beruht auf Diskussionen einer Mailingliste und wurde von David Megginson[6] in eine Spezifikation umgesetzt. Der von ihm verfasste SAX Level 1 unterstützt Elemente, Attribute und PIs. Namensräume und CDATA-Abschnitte werden erst in SAX2 erkannt. Während ein Dokument durchlaufen wird, treten verschiedene Events ein. So zum Beispiel, wenn der Parser ein Starttag gefunden hat. Ebenso tritt ein Event ein, wenn er ein Endtag gefunden hat. Ein XML-Prozessor kann offene Elemente zum

[6] http://www.saxproject.org

Beipiel in einem Stack speichern. Jedes offene Element wird auf den Stack geschoben und jedes schließende Element wird vom Stack entfernt. Somit entspricht dann die Tiefe des Stacks der Verschachtelungstiefe, in der sich der Parser gerade befindet. Eine Methode, die ein Event verarbeitet, heißt Handler.

Bei der Arbeit mit SAX müssen die XML-Dokumente immer wieder durchlaufen werden. Deshalb ist SAX gegenüber dem Document Object Model (Kapitel 2.4.2 auf der nächsten Seite), das ein XML-Dokument im Hauptspeicher als Baum darstellt, relativ langsam.

SAX-Event-Handler Verwendet ein XML-Parser die SAX-Schnittstelle, so liefert er immer ein bestimmtes Ereignis (Event) beim Durchsuchen eines XML-Files. Das erste Event ist `start_document`[7]. Es wird geliefert, wenn der Parser begonnen hat, das Dokument zu verarbeiten. Ebenso wird das letzte Event (`end_document`) geliefert, wenn das Ende des Dokuments erreicht wurde. Dazwischen treten folgende Events auf:

`start_element/end_element`:	Ein Starttag/Endtag wurde gefunden.
`characters`:	Ein String aus Textdaten wurde gefunden.
`processing_instruction`:	Eine PI wurde gelesen.
`comment`:	Ein Kommentar wurde gelesen.
`start_cdata/end_cdata`:	Der Anfang bzw. das Ende einer `CDATA`-Sektion wurde gefunden.
`entity_reference`:	Eine interne Entityreferenz wurde gefunden.

SAX-DTD-Handler Diese Art von Handlern unterstützen DTD-orientierte Events. Das heißt, während die Event-Handler Events aus dem Rumpf- und dem Epilogbereich liefern, werden bei den DTD-Handlern die Events im Prolog ausgelöst. Zum Prolog zählen alle Angaben vor dem Wurzelelement: Dazu zählen die XML-Deklaration, die Dokumenttypdeklaration, usw. Folgende Events können auftreten:

`entity_decl`:	Eine Entitydeklaration wurde gelesen.

[7] Die folgenden Handlernamen werden in der Perl-Implementierung verwendet

`notation_decl`:	Eine Notationdeklaration wurde gelesen.
`unparsed_entity_decl`:	Ein nichtgeparstes Entity wurde gefunden.
`element_decl`:	Eine Elementdeklaration wurde gefunden.
`attlist_decl`:	Eine Attributdeklaration wurde gefunden.
`doctype_decl`:	Eine Dokumenttypdeklaration wurde gefunden.
`xml_decl`:	Eine XML-Deklaration wurde gefunden.

2.4.2 Document Object Model

Neben der Eventverarbeitung kann ein XML-Dokument auch als Baum verarbeitet werden. Das Document Object Model DOM, eine Empfehlung des W3C, bietet dazu die passende Schnittstelle mit einem standardisierten Zugriff auf XML-Dokumente. Sie stellt Funktionen zur Navigation und zur Manipulation des Inhalts und der Struktur zur Verfügung. Wie gezeigt, lässt sich aufgrund der hierarchischen Struktur eines XML-Dokuments dieses sehr gut als Baumgraph darstellen. Das Wurzeltag bildet daher die Wurzel des Baumes. Jedes XML-Element bildet einen Knoten. Jeder Knoten kann weitere Knoten als Kinder besitzen. So sind in Abbildung 2.3 auf der folgenden Seite `name`, `privat` und `business` Kinder von `assistent`. Die Baumstruktur ist deutlich zu erkennen.

Eventorientierte Verarbeitung, wie etwa bei SAX, erfordert das Festhalten vorbeifließender Details. Bei komplexen Strukturen ist dies sehr aufwendig. Baumstrukturen aufzubauen, erfordert allerdings großen Speicheraufwand und teure CPU-Zyklen. So wie es Handler bei der eventorientierten Verarbeitung gibt, gibt es bei diesem Model verschiedene Knotentypen, die in Tabelle 2.1 auf der nächsten Seite zusammengefasst sind.

Elemente, Text, Kommentare usw. werden bei DOM als Knoten aufgefasst. Jeder Markup-Typ (siehe Tabelle 2.1 auf der folgenden Seite) wird durch eine Klasse repräsentiert, die von der Urklasse `Node` abgeleitet ist. Andere Klassen dienen als Container für Knoten. Damit können Teilbäume von einer Stelle zu einer anderen verschoben oder kopiert werden. Speicherverwaltung, Datenstrukturen und Algorithmen zur Verarbeitung eines Baumes sind in DOM nicht spezifiziert, sondern werden dem Programmierer überlassen.

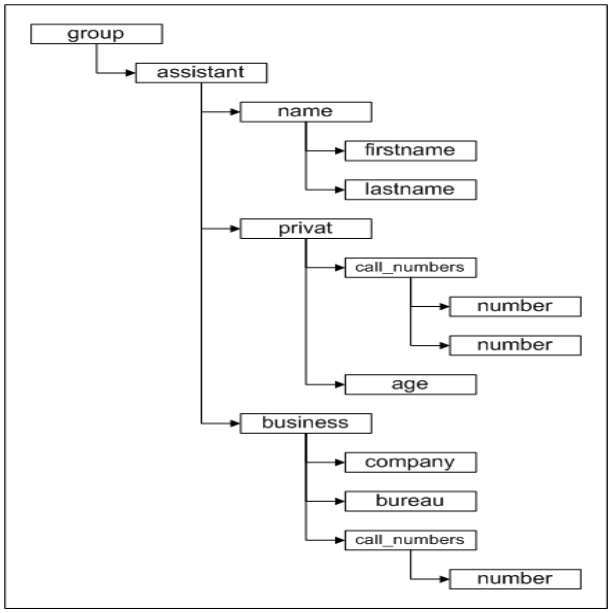


Abbildung 2.3: Darstellung der XML-Datei aus Listing 2.1 auf Seite 10 als Baumdiagramm

Typ	**Eigenschaften**
Element	Name, Attribute und Referenzen auf Kinder
Namespace	Präfix und URI
Character Data	Textstring
Processing Instruction	Target, Daten
`CDATA`-Section	Textstring
Entityreference	Name und Ersetzungstext (oder System-ID und/oder Public-ID)
Kommentar	Textstring

System-ID: der öffentliche Bezeichner des externen Bereichs einer DTD (nur DOM 2)

Public-ID: der Systembezeichner des externen Bereichs einer DTD (nur DOM 2)

Tabelle 2.1: Typische Knotentypen ([RM02])

Vorteile von DOM

Das Document Object Model bietet insgesamt vier Vorteile.

1. Es garantiert eine korrekte Grammatik und die Wohlgeformtheit eines Dokuments, denn durch die abstrakte Repräsentation einer Textdatei als Baum werden offene Elemente und falsch geschachtelte Tags vermieden. Auch der Aufbau falscher Eltern-Kind-Beziehungen wird verhindert.

2. Der Inhalt wird von der Grammatik abstrahiert, indem er im Baum angezeigt wird. Der Baum selbst ist aber nicht an die Grammatik des XML-Dokuments gebunden.

3. Die Bearbeitung von Dokumenten wird vereinfacht, zum Beispiel beim Einfügen neuer Elemente oder beim Löschen aller Elemente eines bestimmten Typs.

4. Die typischen Strukturen einer Datenbank werden widergespiegelt, denn die Art, wie Beziehungen zwischen Elementen beschrieben werden, ist der von hierarchischen und relationalen Datenbanken sehr ähnlich. Somit ist es sehr einfach, mittels DOM Daten zwischen Datenbanken und XML-Dokumenten auszutauschen.

[ABK00] S.159ff

DOM-Spezifikation

DOM Level 1 Dies ist eine Empfehlung des W3C, die 1998 verabschiedet wurde. Anmerkungen der Mitglieder wurden überprüft und eingearbeitet und als Standard für das World Wide Web empfohlen. Ziel dieser Spezifikation ist es, ein sprach- und plattformunabhängiges Interface für XML und HTML-Dokumente zur Verfügung zu stellen. Sie besteht aus zwei Teilen von fundamentalen low-level Funktionen: Core und HTML. Alle Funktionen im Core müssen in jeder DOM-Implementation implementiert sein, die im HTML-Teil enthaltenen Funktionen sind optional. [ABC+98]

DOM Level 2 Dies ist noch keine Empfehlung des W3C, sondern hat den Status einer Candidate Recommendation. In seinen Funktionen baut es auf DOM Level 1 auf. In Zukunft wird DOM Namensräume unterstützen. Es wird möglich sein, Stylesheets zu durchsuchen und zu bearbeiten, den Inhalt von Dokumenten zu filtern, und es wird ein Ereignismodell geben. Außerdem sollen Funktionen zur Manipulation von großen Textblöcken das Arbeiten erleichtern. [LHW+00]

2.5 XML-Dokumente und relationale Datenbanken

Im Electronic Business können relationale Datenbanken nicht alle Anforderungen erfüllen. Dort werden die Daten und ihre Bedeutung getrennt gespeichert. Im Gegensatz dazu befinden sich bei XML alle Informationen in demselben Dokument, d.h. die Daten und deren Bedeutung sind nicht getrennt. Somit kann auf Relationsschemata, Dateibeschreibungstabellen, externe Datentypdefinitionen etc. komplett verzichtet werden. Zwar gibt es bei relationalen Datenbanken XML-Schnittstellen, aber diese sind nur Notlösungen und erreichen nicht die notwendigen Anforderungen bezüglich Datenintegrität und Performance. Mit HTML können zwar die Daten richtig dargestellt werden, aber eine korrekte Verarbeitung kann nicht sichergestellt werden ([Sof03] und [RV03]).

Folgende wichtige Punkte unterscheiden die beiden Lösungen: Bei relationalen Datenbanken hat jede Spalte einen eindeutigen Bezeichner und jeder Eintrag kann durch den *primary key* von den anderen unterschieden werden. XML-Elemente haben keinen eindeutigen Schlüssel. Das ist auch nicht unbedingt notwendig, weil Elemente gleichen Inhalts anhand ihrer Position im Dokument unterschieden werden können. In relationalen Datenbanken haben – nicht wie bei XML – die Zeilen und Spalten einer Tabelle eine feste Reihenfolge. Über die Anordnung der Daten wird nichts ausgesagt. SQL-Strukturen können nicht hierarchisch geschachtelt werden, d.h. es müssen Referenzen zu anderen Tabellen existieren. Ein wichtiger Vorteil von XML ist, dass der Inhalt der Elemente von gemischtem Typ sein darf. So kann zum Beispiel das Element `<book_information>` einmal Zeichen für den Titel und an anderer Stelle Zahlen für die Anzahl der Seiten enthalten.

Kapitel 3

Recherche

3.1 Anforderungen an Querysprachen für XML

Strukturierte und semistrukturierte Datenmodelle stellen nicht die gleichen Anforderungen an eine Anfragesprache. Für semistrukturierte Datenmodelle müssen die Sprachen wesentlich flexibler sein als welche, die zur Anfrage in traditionellen DBMS geeignet sind. Im Allgemeinen ist die Struktur der Daten nicht oder nur zum Teil bekannt. Daher muss so eine Sprache automatische Typumwandlungen beherrschen. Benötigt wird dies zum Beispiel beim Vergleich atomarer Daten verschiedenen Typs (*string* und *integer*, etc.). Semistrukturierte Daten können tief verschachtelt sein. Pfade beschreiben den Weg durch die Hierarchie eines Dokuments. Daher muss die Sprache Mittel zur Verfügung stellen, mit denen die tief verschachtelte Datenstruktur traversiert werden kann. Ein solches Mittel stellen Pfadausdrücke dar. Die Länge dieser Pfade sowie die Strukturtiefe sollten beliebig sein.

Definition 3.1.1 *Unter einem* ***Pfadausdruck*** *ist ein Ausdruck zu verstehen, durch den die Pfade oder Wege in einem Graphen von einem Knoten x zu einem Knoten y beschrieben werden. Ein Pfad ist eine Liste von Knoten, in der aufeinanderfolgende Knoten durch Kanten miteinander verbunden sind [[Sed92]].*

Im Allgemeinen werden Pfadausdrücke mit Hilfe der Kanten notiert. Pfadausdrücke sind immer relativ zu einem Bezugsknoten zu verstehen. Aufeinanderfolgende Kanten werden dabei durch Punkte getrennt.

Für n Knotenname und p Pfadausdruck gilt:

1. Jeder Knotenname n ist ein Pfadausdruck p.
2. Für jeden Pfadausdruck p ist auch $p.p$ ein Pfadausdruck.

In Querysprachen sollte es möglich sein, das Suchergebnis durch Restriktionen zusätzlich einzuschränken. Wünschenswert ist auch eine Funktion, die das Ergebnis einer Suche nach bestimmten Kriterien sortiert darstellt.

Vom W3C wurden außerdem allgemeine Anforderungen an XML-Anfragesprachen gestellt. Diese umfassen, dass eine Anfragesprache für XML mehr als eine Syntax-Bindung haben kann, sowie für den Menschen einfach zu lesen und zu schreiben sein soll. Sie muss außerdem einen deklarativen Charakter haben und darf keine bestimmte Evaluationsstrategie erzwingen. Wird eine XML-Anfragesprache innerhalb von Protokollen verwendet, muss sie unabhängig von diesen sein. Weiterhin sollen Verarbeitungsfehler, die zum Beispiel durch falsche Syntax der Query oder im XML-Dokument entstehen, angezeigt werden. Die Sprache soll aufwärtskompatibel sein, so dass neue Funktionen, die in der Zukunft implementiert werden, auch unterstützt werden.

In Abb. 3.1 auf der nächsten Seite ist die Entwicklung der Anfragesprachen dargestellt.
Dabei gelten folgende Abkürzungen bzw. Bezeichnungen:

XSL:	eXtensible Stylesheet Language
SQL:	Standard Query Language
XQL:	XML Query Language
OQL:	Object Query Language
UnQL:	Unstructered Query Language
XML-QL:	XML-Query Language
Lorel:	Lightweight Object Repository Language
XSLT:	eXtensible Stylesheet Language Transformation
YATL:	Yet Another Tree Language

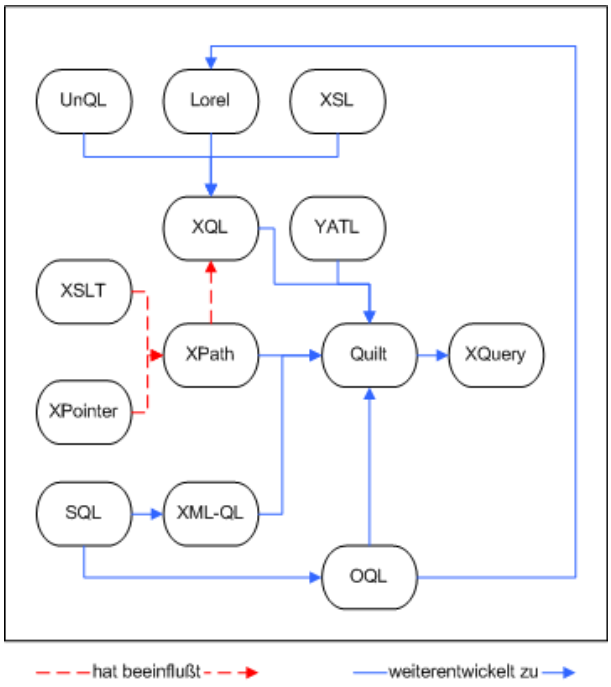


Abbildung 3.1: Anfragesprachen auf einen Blick

3.2 Standard Query Language (SQL)

Historische Entwicklung Die erste Version von SQL stammt aus dem Jahre 1975. Donald D. Chamberlin entwickelte die Structured English Query Language (SEQUEL) als einen Prototyp für relationale Datenbanken. Wenig später folgte SEQUEL2 und wurde kurze Zeit danach in SQL umbenannt. Bisher wurden drei Standards festgelegt:

- 1986: SQL1 oder SQL-86
- 1989: SQL1 + Integrity Enhancement Feature oder SQL-89
- 1992: SQL2 oder SQL-92

Die neueste Version ist SQL3, die bis heute allerdings noch nicht standardisiert wurde. Ursprünglich sollte der Standard 1998 verabschiedet werden. Ebenso existiert bis heute noch kein Standard, der die Umwandlung von XML-Anfragen nach SQL beschreibt. Auch existiert kein Sprachelement mit dem es möglich ist, mit Hilfe einer Art von Pfadausdruck über XML-Daten zu navigieren. Deshalb ist immer noch SQL-92 gültig.

Relationale Datenbanken Die Organisation einer Datenbank nach dem relationalen Modell von E.F. Codd[1] wird als relationale Datenbank bezeichnet. In seinem Modell stellt er zwölf Regeln auf, die eine relationale Datenbank definieren. Somit werden Beziehungen zwischen Daten immer durch zweidimensionale Tabellen dargestellt. Jede Tabelle drückt eine Relation aus. Jede Zeile einer Tabelle repräsentiert ein bestimmtes Objekt. Jede Spalte einer Tabelle besitzt einen Namen, der den Inhalt beschreibt. Der Primärschlüssel einer Tabelle macht jede Zeile eindeutig. Im Idealfall sind auch ohne den Primärschlüssel niemals zwei Zeilen einer Tabelle gleich. Eine Datenbank besteht aus einer Reihe von Tabellen. Ein Datenbanksystem kann mehrere Datenbanken verwalten.

Überblick SQL enthält alle nötigen Sprachbausteine für den Umgang mit einer relationalen Datenbank. Für die Definition von Datenbanken, deren Manipulationen und Zugriffskontrolle stehen drei in SQL integrierte Sprachen zur Verfügung, von denen die ersten beiden in den folgenden Abschnitten näher erläutert werden.

Data Definition Language Eine Datenbank und ihre Tabellen werden mit der Data Definition Language (DDL) definiert und erzeugt. Die DDL stellt die Befehle `CREATE`, `ALTER` und `DROP` bereit. Tabelle 3.1 auf der folgenden Seite stellt einige typische Anweisungen zusammen. Außer Datenbanken und Tabellen können in der DDL auch Indizes, Views, Synonyme und Schemata definiert werden. In Beispiel 3.2.1 wird eine Tabelle mit zwei Spalten erzeugt.

Beispiel 3.2.1

`CREATE TABLE` *Address(*
Name `char`*(30),*
Street `char`*(40),*
City `char`*(25),*
Postcode `integer`*);*

Data Manipulation Language Die ist der umfangreichste Teil von SQL. Mit der Data Manipulation Language (DML) kann auf die Daten in den Tabellen

[1] siehe [Cod03]

Anweisung	**Bedeutung**
`CREATE DATABASE` *datenbankname*	Die Datenbank *datenbankname* und die zugehörigen Systemtabellen werden erzeugt.
`DROP DATABASE` *datenbankname*	Die Datenbank *datenbankname* wird gelöscht.
`CREATE TABLE` *tabellenname(spaltenname datentyp)*	Die Tabelle *tabellenname* mit der Spalte *spaltenname*, deren Werte vom Typ *datentyp* sind, wird erzeugt.
`ALTER TABLE` *tabellenname(...)*	Die Tabelle *tabellenname* wird geändert. In der Klammer kann angegeben werden, was geändert werden soll.
`DROP TABLE` *tabellenname*	Die Tabelle *tabellenname* wird mit allen Spalten und Indizes gelöscht.

Tabelle 3.1: Typische DDL-Anweisungen

einer Datenbank zugegriffen werden. Jede SQL-Query hat den folgenden Aufbau, wobei die in eckigen Klammern stehenden Angaben optional sind:

```
SELECT {*|spaltenname[[AS] alias, ...|(SELECT ...)|ausdruck}
FROM tabellenname[[AS] alias], ...
[ WHERE begingung ]
```

In der `SELECT`-Klausel wird angegeben welche Spalten im Ergebnis einer Suche angezeigt werden sollen. Dies kann durch die Spaltennamen selbst, aber auch durch Konstanten, Funktionen, Mengenfunktionen und Ausdrücke mit arithmetischen Operatoren geschehen. Mit der `alias`-Funktion kann anstatt des Spaltennamens einer Tabelle ein freigewählter Name erscheinen.

In der `FROM`-Klausel werden die Tabellen angegeben, in denen gesucht werden soll. Auch hier kann die `alias`-Funktion angewendet werden. Andere `SELECT`-Klauseln verweisen dann auf diese Tabelle. Weiterhin ist es möglich einen Outer-Join zu definieren, bei dem zwei oder mehr Tabellen miteinander verknüpft werden und eine der anderen Tabelle untergeordnet wird.

Suchkriterien und Join-Bedingungen werden in der `WHERE`-Klausel festgelegt. Hier sind Bedingungen mit relationalen Operatoren, sowie die Funktionen `BETWEEN`, `IN`, `ISNULL`, `LIKE` und `MATCHES` erlaubt.

Das folgende Beispiel (3.2.2) zeigt eine SQL-Query. Die Spalten *Name* und *City* werden aus der Tabelle *Address* ausgewählt. Angezeigt werden die Namen und Städte aller Personen, die in einer Stadt wohnen, die mit `L` beginnt. Tabelle 3.2 zeigt eine Beispieltabelle und Tabelle 3.3 das Ergebnis der Anfrage.

Primärschlüssel	**Name**	**Street**	**City**	**Postcode**
1	Müller	Burgstrasse 5	Leisnig	04987
2	Meier	Gartenweg 23	Leipzig	04318
3	Schulze	Am Bahndamm 1	Berlin	11234

Tabelle 3.2: Die Tabelle *Address*

Beispiel 3.2.2

Anfrage:

SELECT Name,City alias Stadt

FROM Address

WHERE City LIKE 'L%'

Name	**Stadt**
Müller	Leisnig
Meier	Leipzig

Tabelle 3.3: Das Ergebnis aus Beispiel 3.2.2

Durch die Angabe von `ORDER BY` in der `SELECT`-Klausel kann die Ausgabe sortiert werden.

Data Control Language Anweisungen zur Steuerung der Vergabe von Zugriffsrechten stellt die Data Control Language (DCL) zur Verfügung. Benutzern können Systemprivilegien und Rechte auf Datenbankobjekte gewährt bzw. entzogen werden.

Außerdem verfügt SQL über Anweisungen mit denen Einfluss auf die Struktur und Größe der Datenbanken, die Transaktionsverarbeitung und den Datentransfer zwischen Filesystem und Datenbank genommen werden kann. Durch eingebettetes SQL können Anweisungen auch aus den meisten Programmiersprachen heraus formuliert werden. Dazu gehören zum Beispiel Java, C und Perl.

3.3 XML-Query Language (XQL)

Im September 1998 wurde XQL als eine Erweiterung der Syntax der eXtensible Stylesheet Language (XSL) der XSL Working Group des W3C vorgeschlagen. Das Ziel bestand darin, eine deklarative, speziell für XML-Dokumente geeignete Anfragesprache zu entwickeln. Mit dieser Sprache können Knoten und Teilbäume aus einem Dokument extrahiert werden. Das Anfrageergebnis ist wieder ein XML-Dokument.

Datenmodell Unter Verwendung des Document Object Models (DOM) wird ein XML-Dokument als Baum repräsentiert. Die Knoten des DOM-Baumes sind die Elemente eines XML-Dokuments. Attribute sind Datenfelder, auf die von den jeweiligen Elementen aus zugegriffen werden kann. XQL erhält diese Dokumentordnung, d.h. in derselben Reihenfolge, wie Elemente im Originaldokument auftreten, werden sie auch im Ergebnis angezeigt. Somit ist XQL sehr gut geeignet, Fragmente zu finden und zu extrahieren.

Ausdrücke In XQL gibt es Ausdrücke ähnlich der Notation von Pfaden in Dateisystemen. Mit den Anfragequeries kann man durch Angabe von Pfaden durch die Hierarchie des Baumes navigieren. Ein Pfadausdruck hat die Form `path[filter]`, wobei `path` der Ausgabepfad ist. Der Teilbaum des letzten Elements von `path` wird dann im Ergebnis angezeigt. Filter sind bool'sche Ausdrücke, die in eckigen Klammern angegeben werden. Mit `[filter]` läßt sich durch die Angabe von Bedingungen das Ergebnis einschränken. Die Query in Beispiel 3.3.1 auf der folgenden Seite verwendet einen Filter. `/wurzelelement/kind` wird nur ausgegeben, wenn im Baum unterhalb dieses Pfades ein `enkel`-Tag existiert, das den Wert `xyz` hat.

Beispiel 3.3.1

```
/wurzelelement/kind[/wurzelelement/kind/enkel = 'xyz']
```

Die Operatoren AND, OR und NOT werden mit $ gequotet und können beliebig kombiniert werden.

Die Query in Beispiel 3.3.2 liefert die Telefonnummern aller Personen, die Smith heißen und 26 Jahre alt sind.

Beispiel 3.3.2

```
//tel_number[//lastname = 'Smith' $AND$ //age = 26]
```

Die Elemente im Pfad einer XQL-Query entsprechen den XML-Elementen und können im absoluten oder im relativen Kontext zum Wurzelelement des XML-Files stehen, auf das sich die Suche bezieht. Ein Pfad im absoluten Kontext wird immer von einem Slash (/) angeführt. Hierarchisch direkte Nachfolger werden im Pfad durch einen Slash (/) getrennt. Mit Slash-Asterisk-Slash (/*/) wird ein beliebiger Nachfolger und mit einem Doubleslash (//) werden alle Nachfolger eines Elements in die Suche mit einbezogen. So werden in der Query in Beispiel 3.3.3 alle Tags `<lastname>` selektiert, die Enkel des Tags `<assistant>` sind. Auf die Namen von Attributen kann man mit dem @-Zeichen zugreifen (Bsp. 3.3.4).

Beispiel 3.3.3

```
/group/assistant/*/lastname
```

Beispiel 3.3.4

```
/wurzelelement/kind1[\@kind2 = 'attr']
```

Durch Verwendung des Doppelslashs werden mit `//number` alle Tags mit dem Namen `<number>` selektiert. Wie auch in XPath[2] kann in XQL-Anfragen auf die Reihenfolge von Elementen Bezug genommen werden. Dafür wird die Funktion `index()` zur Verfügung gestellt. Den Zugriff auf das erste Kind eines Elementes zeigt Beispiel 3.3.5 auf der nächsten Seite. Es werden auch verschiedene Typen von Knoten unterschieden – Attribute, Elemente und Kommentare. Mit `nodeType()` kann der Typ eines Knotens ermittelt werden.

[2] siehe Kapitel 3.5.5 auf Seite 41

Beispiel 3.3.5

```
//call_numbers/number[index()=0]
```

XQL selbst bietet keine Möglichkeit, das Ergebnis sortiert auszugeben. Dazu muss ein entsprechendes Hilfsprogramm verwendet werden. Als eine reine Anfragesprache bietet sie keine Konstrukte zur Manipulation eines XML-Dokuments. Mit XQL können also keine Änderungen in den XML-Dokumenten durch löschen, ändern oder hinzufügen von Elementen und Dateninhalten vorgenommen werden. Die Pfadausdrücke sind keine generellen Pfadausdrücke (vgl. entsprechende Definition in Abschnitt 3.5.2 auf Seite 40), denn es existiert kein Operator für die Kleensche Hülle, sowie kein Auswahloperator (vgl. [KM03]).

[Sta02], [KM03]

3.4 XQL versus SQL

Beide Anfragesprachen sind in gewisser Hinsicht ähnlich, denn sie sind ein Mittel, eine Datenbank nach bestimmten Kriterien zu durchsuchen. Der Grundaufbau der Syntax beider Sprachen ist fast identisch. Das "SELECT FROM WHERE" von SQL findet sich auch in XQL wieder, nur in anderer Form. Dort wird zuerst der auszugebende Knoten angegeben, dann folgen die Bedingungen und zum Schluß werden alle Dateien genannt, die durchsucht werden sollen. Abbildung 3.2 auf der nächsten Seite verdeutlicht die Parallelen beider Sprachen. In der Anwendung beider Anfragesprachen gibt es aber erhebliche Unterschiede. Mit SQL werden relationale Datenbanken durchsucht, die in einem DBMS gespeichert sind. XQL läßt sich bereits auf ein XML-File anwenden.

3.5 Weitere Anfragesprachen

3.5.1 XQuery

XQuery ist ein Sprachvorschlag der W3C XML Query Group. Sie geht aus der Anfragesprache *Quilt* hervor (siehe auch Kapitel 3.5.6 auf Seite 44), die wiederum durch die Sprachen SQL, Lorel, XML-QL, YATL u.a. beeinflußt wurde (siehe Abb. 3.1 auf Seite 27). Durch XQL wurde sie mitgeformt. Daher ist die Syntax

Ausgabepfad	Filterbedingungen	XML-File(s)
/element1/element2	[/element1/element2 = 500 and /element1/element3 < 2000]	xmlfile.xml

SELECT	/element1/element2
FROM	xmlfile.xml
WHERE	[/element1/element2 = 500 and /element1/element3 < 2000]

Abbildung 3.2: XQL und SQL im Vergleich

von XQuery sehr umfangreich. Dazu gehören Pfadausdrücke, Elementkonstruktoren, FLWR-Ausdrücke[3], Operatoren, Bedingungen, Quantoren, Funktionen und Ausdrücke, mit denen Datentypen getestet und modifiziert werden können. Die Syntax der Pfadausdrücke ist dieselbe wie bei XPath. Ein XML-Dokument wird als Baum betrachtet, dessen Wurzel das Wurzeltag des Dokuments ist. Mit XQuery ist es auch möglich, XML-Elemente zu erzeugen. Dadurch kann das Ergebnis wieder als XML-Dokument ausgegeben werden. Dies geschieht durch indirektes Einbinden eines XML-Ausdrucks in eine Query.

Sequenzen Der XQuery-Vorschlag erweitert das Datenmodell von XPath 1.0 um einige Punkte. Es können einzelne XML-Dokumente, wohlgeformte Fragmente und Sequenzen von Dokumenten dargestellt werden. Eine Sequenz ist eine geordnete Menge von Knoten. Knoten können selbst geschachtelte Knoten enthalten. Daher kann eine Sequenz eine geordnete Menge von Bäumen darstellen. Sequenzen selbst können keine weiteren Sequenzen enthalten, also existieren sie nur auf der obersten Ebene. Allerdings sind in Sequenzen Duplikate erlaubt (Abb. 3.3 auf der folgenden Seite).

[3]FLWR = FOR, LET, WHERE, RETURN

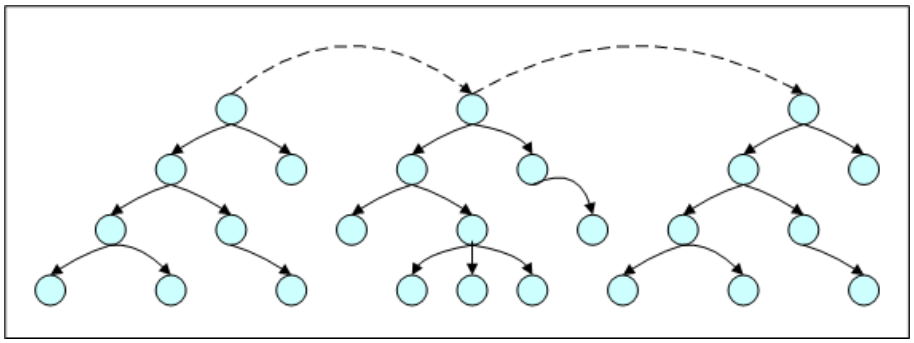


Abbildung 3.3: Sequenzen in XQuery

Ausdrücke allgemein Ausdrücke sind die Grundbausteine von XQuery. Sie können beliebig ineinander verschachtelt werden. Es existieren verschiedene Formen von Ausdrücken. So dienen die Elementkonstruktoren zur Erstellung oder Ableitung neuer XML-Elemente. Die aus XPath bekannten Pfadausdrücke dienen zur Selektion von Dokumentbestandteilen. Ausdrücke können auch datentypspezifische Operatoren und Funktionsaufrufe von Standardfunktionen oder selbst definierten Funktionen enthalten. FLWR-Ausdrücke erlauben die Stellung ähnlicher Queries wie in SQL. Zur Steuerung der Auswertung können die Anweisungen Bedingungen enthalten. Mit den Quantoren `ANY` und `ALL` lassen sich die Ausdrücke quantifizieren. Es ist auch möglich auf Datentypen oder Typumwandlung zu testen.

Einfache Ausdrücke Einfache numerische Werte und Zeichenketten werden durch Konstanten und Literale ausgedrückt. Konstanten können vom Typ *Integer* (`{-- xs:integer --}`), *Decimal* (`{-- xs:decimal --}`) oder *Double* (`{-- xs:double --}`) sein. Literale stehen in einfachen ('Zeichenkette') oder doppelten ("Zeichenkette") Hochkommata und sind vom Typ String (`{-- xs:string --}`).

Beispiel 3.5.1

```
LET $assistant := //assistant/name[lastname = 'Smith']
RETURN $assistant
```

Arithmetische Ausdrücke werden unter der Verwendung folgender Operatoren gebildet: $+$, $-$, $*$, *mod* und *div*. Mit den runden Klammern (und) können Teilausdrücke geklammert werden und somit kann auch die Auswertungsreihenfolge beeinflußt werden.

Mit Vergleichsausdrücken lassen sich zwei Operanden vergleichen. Es werden vier verschiedene Arten unterschieden:

- Allgemeine Vergleichsoperatoren mit Sequenzen: $<$, $<=$, $>$, $>=$, $=$ und $!=$
- Einfache Wertevergleichsoperatoren: gleich: EQ, ungleich: NE, kleiner: LT, kleiner gleich: LE, größer: GT, größer gleich: GE
- Knotenidentitätsvergleiche: identisch: $==$, nicht identisch: $!==$
- Abfolgevergleich von Knoten bezüglich der Dokumentordnung und in Bezug auf die Beziehung zum Vorgänger (`PRECEDES`) und Nachfolger (`FOLLOWS`)[4]: $<<$ und $>>$

Weiterhin gibt es logische Operatoren (`AND` und `OR`), die *wahr* oder *falsch* liefern. Mit `NOT` kann das Ergebnis negiert werden.

Funktionsaufrufe liefern einen Wert, dessen Typ von der verwendeten Funktion festgelegt wird. Dort wo der Typ des Funktionswertes eingesetzt werden kann, kann auch ein Funktionsaufruf stehen. Außer den vordefinierten Funktionen lassen sich auch selbst definierte einsetzen. Jeder Funktion können entsprechend vorher deklarierte Parameter übergeben werden. Kommentare innerhalb einer Anfrage beginnen mit `{--` und enden mit `--}`.

Das Ergebnis einer Anfrage kann mit dem `SORT BY`-Kommando nach bestimmten Kriterien sortiert werden. Mit der Angabe von `ASCENDING` kann das Ergebnis beispielweise aufsteigend geordnet werden. XQuery erlaubt auch die Verwendung des Existenz- und des Allquantors. Mit dem `IF - THEN - ELSE` - Konstrukt lassen sich Bedingungen in eine Anfrage einbetten. Der Standard von XQuery umfaßt sieben Dokumente. Sie alle haben den Status *Working Draft*. [KM03]

[4]PRECEDES und FOLLOWS entfallen im XQuery Draft vom August 2002

3.5.2 XML-QL

Die XML-Suchsprache *XML-QL* wurde von den *AT&T Labs* entwickelt und auch implementiert. Das Konzept und die Syntax sind stark an SQL angelehnt. Mit dem graphenbasierten Datenmodell ist XML-QL sowohl Anfragesprache als auch Transformationssprache. Es ist möglich Join-Operationen anzuwenden, sowie Daten aus XML-Files zu extrahieren und daraus neue XML-Files zu generieren. Außerdem werden geordnete und ungeordnete Sichten auf ein XML-File unterstützt. Die Rückgabe der Ergebnisse erfolgt in Tupel-Mengen.

Datenmodell Ähnlich wie Lorel (Abschnitt 3.5.4 auf Seite 39) benutzt XML-QL ein graphenbasiertes Datenmodell. Die Elemente eines XML-Dokuments werden auf einen Baum abgebildet. Jedes Element entspricht einem Knoten im Baum. Subelementbeziehungen werden durch Kanten ausgedrückt. Existieren `ID/IDREF`-Beziehungen im Baum, wird aus diesem ein Graph. Elemente mit Attributen vom Typ `IDREF` werden mit den entsprechenden Elementen (die im XML-Dokument ein Attribut vom Typ `ID` besitzen) über zusätzliche Kanten verbunden, die referenziert werden. Der so entstandene Graph ist gerichtet. `#PCDATA`-Abschnitte werden wie andere Elemente behandelt und als Knoten dargestellt. Attribute werden unmittelbar an einem Element mit abgelegt. Durch die Referenzen können mehrere Pfade von der Wurzel zu einem bestimmten Knoten existieren. Deswegen muss die Suchlogik sicherstellen, dass ein Knoten jeweils nur einmal besucht wird.

Ausdrücke Durch die Anlehnung an SQL sind auch die Ausdrücke SQL-ähnlich. Das Ergebnis einer Anfrage hat Baumstruktur. Mit XML-QL ist es möglich neue Elemente zu erzeugen. Der `IN`-Operator bietet Zugriff aus verschiedenen Dokumenten. XML-QL bietet Operationen, mit denen ein XML-Dokument restrukturiert werden kann. Jede Anfrage besteht aus einer `WHERE`-Klausel und einer `CONSTRUCT`-Klausel.

Beispiel 3.5.2

```
WHERE pattern IN source ...
CONSTRUCT output
```

`pattern` sind Muster über die Graphenstruktur, `source` der Uniform Resource Identifier (URI) eines Dokuments oder eine Variable, `output` definiert die Struktur der Ausgabe. In der `WHERE`-Klausel verwendete Variablen haben denselben Wert, wie bei Verwendung der `CONSTRUCT`-Klausel. Die `CONSTRUCT`-Klausel bildet einen Teilbaum, der dem Ergebnis hinzugefügt wird. Enthält eine Variable weitere Unterknoten, werden diese ebenfalls in das Ergebnis eingesetzt. Somit repräsentiert eine Variable nicht nur elementare Werte, sondern auch Subgraphen. Auf diese Weise werden Gruppierungen und Verbünde möglich. Dadurch geht allerdings die Struktur der Dokumente verloren.

Beispiel 3.5.3

Beispielanfrage in XML-QL.

```
WHERE
<name>
   <firstname>$fn</first_name>
   <lastname>$ln</last_name>
</name>
IN personal.xml
CONSTRUCT
<letter>
   <firstname>$fn</first_name>
   <lastname>$ln</last_name>
</letter>
```

XML-QL unterstützt alle Grundoperationen, die die Anforderungen an eine XML-Anfragesprache stellen, sowie alle Konstrukte, die das XML-Datenmodell zur Verfügung stellt. Ein Nachteil von XML-QL ist, dass durch die Referenzen die Reihenfolge der Ergebnismenge nicht unbedingt der des Originaldokuments entspricht. Außerdem wird die Struktur des Dokuments im Ergebnis nicht angezeigt. Das heißt, alle Kindelemente müssen ausdrücklich mit angegeben werden, damit sie ausgegeben werden. Dadurch werden die Anfragen sehr lang. Mit der `ORDER BY`-Klausel kann das Ergebnis nach den angegebenen Kriterien sortiert werden. ([KM03] und [FSW99])

3.5.3 Object Query Language (OQL)

Für objektorientierte Datenbanken wurde 1993 OQL von der Object Database Management Group (ODMG) vorgestellt. Die Anfragen sind wie bei SQL in der `select ... from ... where` Form aufgebaut. Die *Selectklausel* gibt an, welche Werte im Anfragergebnis enthalten sein sollen. Die Menge der Objekte wird in der *Fromklausel* definiert. Die *Whereklausel* enthält Restriktionen, die das Suchergebnis einschränken.

Beispiel 3.5.4

```
select X.name
from X in cities
where X.olympia = "yes"
```

Mit der Anweisung aus Beispiel 3.5.4 werden die Namen der Städte ausgegeben, in denen schon einmal Olympische Spiele stattgefunden haben. Durch Beziehungen ist es möglich, in OQL von einem Objekt zu einem anderen zu navigieren. Diese Navigation wird durch Pfadausdrücke realisiert. Die Pfade werden in der Form `<object>(.<property>)+` notiert. `<property>` kann sowohl ein Attribut eines Objekts sein sowie seine Beziehung zu einem anderen Objekt. Das + hat dieselbe Bedeutung wie bei den Regular Expressions in Kapitel 4.6.2 auf Seite 59, der Klammerausdruck vor dem Plus-Zeichen darf beliebig oft, muss aber mindestens einmal vorkommen.

OQL ist eine Anfragesprache für ein strukturiertes Datenmodell. Anforderungen, die ein semistrukturiertes Datenmodell stellt, werden nicht erfüllt. [Sta02]

3.5.4 Lorel

Lightweight Object Repository Language (Lorel) ist die Anfragesprache des Lightweight Object Repository (Lore) DBMS, das an der *Stanford University* entwickelt wurde. Es ist eine Erweiterung der OQL[5]. Hauptaufgabe des Systems ist die effiziente Speicherung, Anfrage und das Updaten von semistrukturierten Daten. Ein wichtiges Konzept ist die automatische Typumwandlung bei Vergleichen.

[5] siehe Kapitel 3.5.3

Die Anfragen in Lorel entsprechen der Form *select ... from ... where*. Lorel liefert auch bei Dokumenten unbekannter Struktur verwertbare Ergebnisse.

Datenmodell In Lorel existieren einfache Pfadausdrücke und generelle Pfadausdrücke, die auf regulären Ausdrücken basieren[6].

Definition 3.5.1 *Ein einfacher Pfadausdruck ist eine Folge $Z.l_1 \dots l_n$, wobei Z für den Namen eines Objekts oder einer Variablen, die ein Objekt bezeichnet, steht und $l_1 \dots l_n$ die beschrifteten Kanten eines Graphes sind.*
Ein Datenpfad ist eine Folge $o_0 l_1 o_1 \dots l_n o_n$, wobei Objekte mit o_i und Kanten zwischen zwei Objekten o_{i-1} und o_i mit l_i bezeichnet werden [AQM+97].

Beispiel 3.5.5
Ein Pfadausdruck basierend auf Abb. 2.3 auf Seite 22.

```
group.assistant.name.firstname
```

Definition 3.5.2 *Ein genereller Pfadausdruck besteht aus einem Objektnamen, gefolgt von mehreren Komponenten. Für die Komponenten gilt:*

1. *sofern l ein Bezeichner ist, dann ist .l eine Komponente eines generellen Pfadausdrucks.*

2. *Wenn p_1 und p_2 Komponenten eines generellen Pfadausdrucks sind, dann sind auch $p_1 p_2$, $p_1|p_2$, (p_1), $(p_1)?$, $(p_1)+$ und $(p_1)*$ Komponenten eines generellen Pfadausdrucks.*

[AQM+97].

Lorel unterstützt auch Graphen, die Zyklen enthalten. Der Quantor * kann in bestimmten Fällen unendlich viele Ergebnisse liefern. Lorel verhindert dies, indem ein Pfadausdruck maximal einmal durchlaufen werden darf. Die Zeichen % und # werden in Lorel als Wildcards genutzt. So steht % für Null und mehrere Zeichen innerhalb eines Bezeichners. Mit # können Datenpfade beliebiger Länge

[6] siehe Kapitel 4.6.2 auf Seite 59

abgekürzt werden. Ist die Struktur eines Dokuments nicht vollständig bekannt, ist es trotzdem möglich, Queries zu bilden. Weiterhin können Namen für Pfadausdrücke vergeben werden, die dann in anderen Pfadausdrücken verwendet werden können.

Generelle Pfadausdrücke sind ein mächtiges Konstrukt. Nicht nur einzelne Knoten, sondern auch Mengen von Knoten können so ausgewählt werden. Wie bei OQL kann das Ergebnis mit Hilfe der Whereklausel eingegrenzt werden. Mit `sort by` steht eine Funktion zur Verfügung, mit dem das Ergebnis nach bestimmten Kriterien sortiert werden kann. Lorel stellt außerdem Funktionen bereit, mit denen Objekte in einem Graphen geändert oder neu eingefügt werden können. Zum Löschen von Objekten dagegen existieren keine speziellen Operationen. Ist ein Objekt im Graphen nicht mehr erreichbar, gilt es als gelöscht. Das Löschen kann somit erreicht werden, wenn alle Kanten, die zu diesem Objekt führen, gelöscht werden. Bei sehr vielen Kanten ist dieser Vorgang sehr ineffizient. [Sta02], [FSW99]

3.5.5 XML Path Language (XPath)

Im November 1999 wurde XPath vom W3C vorgestellt. Die Anfragesprache stellt eine einheitliche Syntax und Semantik für die Funktionalität der eXtensible Stylesheet Language Transformation (XSLT) und von XPointer bereit. Mit XSLT lassen sich XML-Dokumente von einem Format in ein anderes transformieren. Mit XPointer können Verweise innerhalb von XML-Dokumenten hergestellt werden. So ist es möglich, mit XPath Teile eines XML-Dokuments zu adressieren. Mit XPath werden Funktionen zur Manipulation von Zeichenketten, Zahlen und boolschen Werten, sowie zur Navigation in der Struktur der Dokumente zur Verfügung gestellt. Die Notation der Navigation ist ähnlich der Internet-URLs.

Datenmodell Wie bei einigen anderen Anfragesprachen wird die Struktur eines XML-Dokuments als Baum erkannt. Einzelne Elemente werden Knoten genannt, zusammenhängende Gruppen Knotenmenge.

XPath unterscheidet verschiedene Knoten in einem Baum:

1. Jeder Baum hat einen *Wurzelknoten.*

Der Wurzelknoten eines XML-Dokuments ist mit diesem nicht identisch. Er ist ein Nachfolgeknoten des Wurzelknotens des Baumes.

2. Jeder *Elementknoten* steht für ein XML-Element.
   Elementknoten können weitere Elementknoten, Attributknoten, Namensraumknoten oder Textknoten[7] als Kinder haben.

3. Die Blätter eines Baumes heißen *Textknoten.*
   In ihnen steht der Inhalt der XML-Elemente.

4. *Attributknoten* sind keine Kinder von Elementknoten, sondern ihnen zugeordnet.

5. *Kommentarknoten*

6. *Verarbeitungsanweisungsknoten*

7. *Namensraumknoten* für XML-Dokumente, die mehrere DTDs benutzen.

Das zentrale Element in XPath ist der auszuwertende *Ausdruck.* Im Ergebnis erhält man eine Knotenmenge, einen boolschen Wert, eine Zahl oder eine Zeichenkette. Innerhalb eines XML-Dokuments wird mit einem *location path* navigiert. Einzelne Navigationsschritte werden durch / getrennt, absolute Pfade beginnen immer mit / und beziehen sich auf das Wurzelelement. Jeder Schritt liefert eine Menge von Knoten. Diese gelten für den folgenden Navigationsschritt als Kontext. Zum Kontext gehören der aktuelle Knoten, dessen Position innerhalb des Kontextes selbst, die Größe des Kontextes (Anzahl der Knoten), die in einer Bibliothek verfügbaren Funktionen, wirksame Variablenbindungen und Namensraumangaben. Ein Navigationsschritt besteht aus drei Teilen – einer *Achse*, einem *Knotentest* und *Prädikaten.*

Achsen Eine Achse beschreibt die Beziehung zwischen den gesuchten Knoten und dem Kontextknoten. Tabelle B.1 auf Seite 112 enthält eine Liste der definierten Achsen in XPath. Die Bedeutung ausgewählter Achsen ist in Abbildung 3.4 auf Seite 44 dargestellt.

[7] enthalten den #PCDATA-Inhalt

Knotentest Die Syntax sieht eine Trennung zwischen Achse und Knotentest durch einen doppelten Doppelpunkt (::) vor (`child::para[position()=1]`). Der Knotentest kann entweder aus einem XML-Element bestehen, dem Wildcard * oder einer Testfunktion für einen bestimmten Knotentyp. So werden Knotentyp und Namen der gesuchten Knoten spezifiziert.

Beispiel 3.5.6

Der Ausdruck `child::text()` *liefert nur Textknoten.*

Mögliche Eigenschaften für den Knotentest und Beispiele von Ausdrücken befinden sich in den Tabellen B.2 auf Seite 112 und B.3 auf Seite 113.

Prädikate Durch Angabe der optionalen, in eckigen Klammern ([ ]) stehenden, Prädikate kann die Auswahl der Knoten weiter eingeschränkt werden. Funktional sind sie der `SELECT`-Klausel in SQL ähnlich. Es werden verschiedene Operatoren unterstützt:

- Die logischen Operatoren `AND` und `OR`, die *wahr* oder *falsch* liefern.
- Die Vergleichsoperatoren $<$, $<=$, $>$, $>=$, $=$ und $!=$, die auch *wahr* oder *falsch* liefern.
- Die Operatoren $+$, $-$, $*$, *mod* und *div* werden auf numerische Werte angewendet und liefern ebenfalls einen numerischen Wert zurück.
- Mit dem Operator | werden Knotenmengen vereinigt. Er liefert selbst eine Knotenmenge zurück.
- Mit (und) können Teilausdrücke geklammert werden. Hiermit kann auch die Auswertungsreihenfolge beeinflußt werden.

Boolsche Werte, unendliche sowie nicht definierte numerische Werte werden in XPath durch bestimmte Zeichenketten ausgedrückt. Demnach stellen "true" und "false" die boolschen Werte *wahr* und *falsch* dar. Nicht definierte Zahlen werden in NaN[8], Null in 0, und unendliche Werte in $Infinitiy$ für $+\infty$ und $-Infinitiy$ für $-\infty$ konvertiert.

[8]Englisch: Not a Number

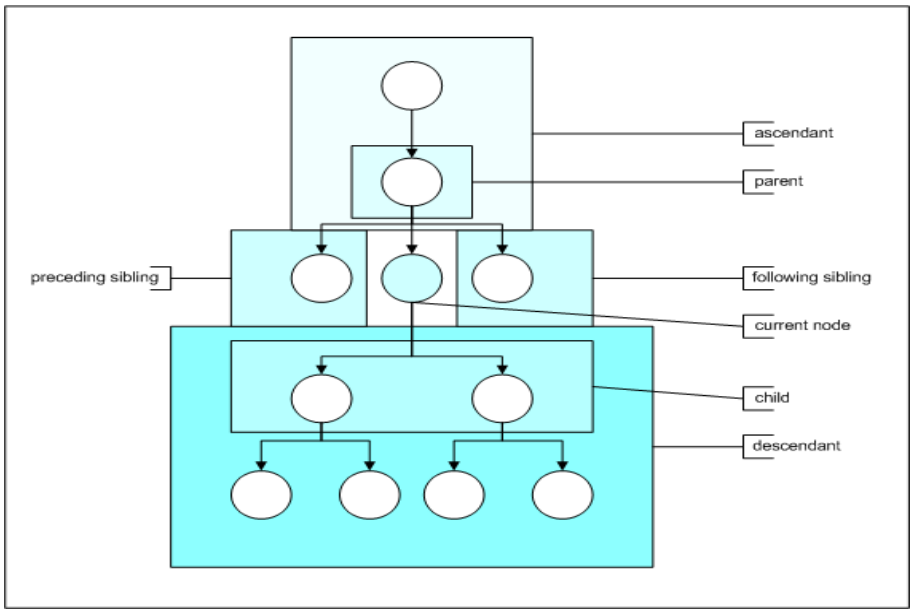


Abbildung 3.4: Achsen bei XPath

Bei der Auswahl von Teilmengen von Knoten aus einem XML-Baum ist es nicht nötig, die exakte Tiefe eines Elements zu kennen. Durch die Achsen `descendant` und `ancestor` kann die Suche ausgeweitet werden. XPath besitzt keine Funktionen, mit denen der Baum manipuliert werden kann. Das Einfügen, Ändern oder Löschen von Elementen ist daher nicht durchführbar, ([Sta02], [KM03]).

3.5.6 Quilt

In die Entwicklung von *Quilt* gingen Erkenntnisse und Erfahrungen bereits bestehender Sprachen, wie z.B. XML-QL, XQL, YATL und OQL ein. Aus XQL (vgl. Kap. 3.3 auf Seite 31) und XPath (vgl. Kap. 3.5.5 auf Seite 41) wurde zum Beispiel die Syntax (Pfadausdrücke) zur Suche in hierarchischen Dokumenten übernommen. Die Kombination von Klauseln stammt aus SQL (vgl. Kap. 3.2 auf Seite 27). Die Suche kann in einzelnen XML-Dokumenten, XML-Dokumentteilen und Sammlungen von XML-Dokumenten erfolgen. Diese drei Typen können auch das Ergebnis von Anfragen sein. Wie in XPath bestehen Pfadausdrücke aus mehreren Schritten. Jeder Schritt repräsentiert eine Bewegung durch das Dokument. Das Ergebnis des aktuellen Schritts dient gleichzeitig als Ausgangspunkt für den nächsten Schritt. Zur Herausfilterung von Elementen werden Prädikate benutzt.

Sie werden in jedem Schritt angewendet. Die in Quilt definierten Pfadoperationen sind in Tabelle B.4 auf Seite 114 zusammengefaßt.

Beispiel 3.5.7

Eine Query in Quilt:

```
document("personal.xml")/group//name[firstname = "John"]
```

Mit Hilfe von Konstruktoren können beliebige XML-Elemente erzeugt werden. Ein Liste von Ausdrücken wird von einem Anfangs- und einem Endetag umschlossen. Die Liste spezifiziert den Inhalt des Elements. Das Anfangstag darf ein Attribut enthalten. Als Elementinhalt dienen Variable, die an anderen Stellen in einer Anfrage gebunden sind.

Beispiel 3.5.8

Konstruktor in Quilt

```
<constr attr = $id>
   <name> $name </name>
   <number> $number </number>
</constr>
```

Mit Hilfe der Konstruktoren kann so das Format der Ausgabe beeinflußt werden.

Die meisten Anfragen werden mit dem FLWR-Konstrukt erstellt. Dabei werden im `FOR`-Teil Variablen als Iteratoren über Listen von Tupeln gebunden. Im `LET`-Teil werden Variablen an Ausdrücke gebunden. In der `WHERE`-Klausel werden die erzeugten Tupel gefiltert. Mehrere Filter können durch die logischen Operatoren `AND`, `OR` und `NOT` miteinander verbunden werden. In der `RETURN`-Klausel wird das Format des Ergebnisses definiert, z.B. mit Hilfe der Konstruktoren.

Beispiel 3.5.9

Anfrage in Quilt

```
FOR $i IN document("personal.xml")//lastname
LET $a := min(document("personal.xml")//assistant[//age < 30])
```

```
RETURN
   <assistant>
   <name> $i </name>
      <age> $a </age>
</constr>
```

In Quilt lassen sich Dokumente auf bestimmte Elemente reduzieren. Dies geschieht mit Hilfe von `FILTER`-Ausdrücken. Die hierarchische Struktur bleibt erhalten, aber alle unerwünschten Elemente werden entfernt. Der `FILTER` hat zwei Operanden. Der erste Operand liefert alle Elemente eines XML-Dokuments. Mit dem zweiten Operanden wird bestimmt, welche der Elemente erhalten werden sollen. Alle anderen Elemente werden gelöscht. Die Liste der Elemente wird mit `|` getrennt.

Beispiel 3.5.10

Ein `FILTER`*-Ausdruck*

```
<telefonverzeichnis>
   document("personal.xml") FILTER
      //name | //call_numbers
</telefonverzeichnis>
```

Außer mit den bisher genannten Funktionen das Format des Ergebnisses zu beeinflussen, ist es außerdem möglich, mit dem `SORT BY` Kommando die Ausgabe zu sortieren. Mit den Konstruktoren lassen sich neue XML-Elemente erstellen. XML-Dokumente können aber in keiner Weise verändert werden, weder durch Einfügen, Ändern oder Löschen von Elementen.
[FSW99], [Sta02] und [CRF00]

3.5.7 YATL

Die Yet Another Tree Language (YATL) basiert auf dem YAT Datenmodell für die Beschreibung von Ein- und Ausgabedaten und wurde entwickelt, um den Datenaustausch zwischen heterogenen Datenquellen zu spezifizieren. Sie ist eine

der ersten Datenbanksprachen, welche Querying, Konvertierung und Integration unterstützt. Die Anfragen in *YATL* entsprechen der Form *"make ... match ... where"*. [FSW99]

3.5.8 XML-GL

Die *XML Graphical Language* (XML-GL) unterscheidet sich von den anderen XML-Suchsprachen aufgrund der Tatsache, daß sie auf der graphischen Repräsentation in Form von Graphen von XML-Files und DTD's aufbaut. Da Anfrage und Ergebnisse auf die gleiche Weise aufgebaut sind, stellt sich die Schnittstelle sehr benutzerfreundlich dar. Die Anfrage wird durch ein Graphenpaar beschrieben, von dem das eine für die Suche und das andere für die Formatierung der Ergebnisse verantwortlich ist. [FSW99]

3.6 Alternative Ansätze

3.6.1 Glimpse

Überblick *Glimpse* ist ein Tool, mit dem man ganze Filesysteme durchsuchen kann. Dabei stellt es eine Kombination aus sogenannten *grep-like*-Tools und *index-based*-Tools dar. Hauptmerkmal von *grep-like*-Tools ist die fehlertolerante Suche. *index-based*-Tools haben einen Index über die Originaldaten, der unter Umständen größer sein kann als die Originaldaten selbst[9]. *Glimpse* unterstützt die fehlertolerante Suche und hat einen sehr kleinen Index. Die Struktur der Daten ist für die Suche nicht relevant, da immer die Volltextsuche angewandt wird. Durch boolsche Operatoren ist die Aufstellung von Queries sehr flexibel.

Arbeitsweise Als Grundprogramm wird *agrep* benutzt, das eine fehlertolerante Suche mit Angabe der Anzahl der zu tolerierenden Fehler bis hin zu *best match* erlaubt. Der gesamte Datenbestand wird in 2^8 Blöcke zerlegt. In der Indextabelle wird jedes Wort genau einmal zusammen mit Verweisen auf die Blöcke, in denen das Wort ein oder mehrmals auftaucht, abgespeichert. Ein Verweis enspricht ei-

[9] Bei *Information-Retrieval-Systeme* werden *inverted indexes* benutzt. Diese können 50% bis 300% der Textgröße ausmachen. Der Index von *Glimpse* beträgt nur 2% bis 4% des Texts.

nem Byte, woraus der kompakte Index resultiert. Zuerst wird im Index mit *agrep* die Liste der Blöcke ermittelt, und danach findet eine sequentielle Suche in jedem einzelnen Block (wieder mit *agrep*) statt.

Vorteile Der sehr kleine, leicht zu modifizierende und schnell erzeugbare Index ist ein großer Vorteil. Durch die fehlertolerante Suche werden auch durch Rechtschreibfehler entstellte Texte gefunden. *Glimpse* ist leicht anpaßbar auf einen parallelen Computer. Queries können Wildcards, Buchstabenklassen[10] und Regular Expressions enthalten.

Nachteile Die Suchzeiten sind größer als bei Programmen, die einen größeren Index benutzen. Das Programm ist eindeutig zu langsam, wenn in Queries mit logischen Ausdrücken (AND, OR) gearbeitet wird, und wenn in der Sprache häufige Wörter[11] vorkommen.

3.6.2 Tamino

Tamino steht für Transactional Architecture for Managing Internet Objects. Entwickelt von der *Software AG*, gehört *Tamino* zu den nativen XML-Datenbankservern, die die Unzulänglichkeiten relationaler DBMS beseitigen sollen und alle unter XML realisierbaren Möglichkeiten ausreizen. Die Hauptmerkmale von *Tamino* sind die native Speicherung der Daten sowie die Anfragesprache *X-Query* (vgl. Kapitel 3.5.1 auf Seite 33), mit der hierarchisch angeordnete XML-Daten aus Dokumenten extrahiert werden können. Während die gängigen Datenbankhersteller wie IBM, Oracle und Microsoft XML-Erweiterungen für ihre Produkte anbieten, die meist nur für daten-zentrische Dokumente[12] ausgerichtet sind und für Anfragen von dokument-zentrischen Dokumenten[13] Volltextsuche anwenden,

[10] z.B. [A-D], [e-z]

[11] z.B. in, und, am, ...

[12] Kriterien für Datenzentrik:

Gleichmäßige reguläre, nicht tief verschachtelte Struktur

Daten lassen sich in kleine Datenblöcke aufspalten, die als Ganzes betrachtet werden

[13] Kriterien ür Dokumentzentrik:

Unregelmäßige Struktur, Verschachtelung beliebiger Tiefe, gemischter Inhalt

Folge der Elemente ist signifikant

ist *Tamino* genau dafür entwickelt worden.

Architektur Das Herzstück von *Tamino* ist die *X-Machine*. Ihre Aufgabe ist das Auffinden und Speichern von Daten im XML-Format. Für die Datenablage und das Absetzen einer Query ist das *Data Dictionary* zuständig. Über die *Data Map* erfährt es alle Metainformationen der zulässigen Schemata. In relationalen DBMS wird jede Tabelle nach einem zuvor definierten Schema aufgebaut. Bei *Tamino* existiert zwar auch ein solches Konzept. Es ist aber nicht zwingend. Tatsächlich lassen sich alle *Document Type Definitions/Schemes* in jedes beliebige Format konvertieren. So ein Schema enthält auch Informationen zur Schaffung von Index-Strukturen, mit deren Hilfe die Organisation der Daten optimiert und die Suche beschleunigt werden kann.

Vorteile Außer der nativen XML-Engine wurde auch eine native SQL-Engine eingebaut, so daß auch relationale und unter Umständen objektrelationale Daten direkt in *Tamino* abgelegt, analysiert und abgefragt werden können. Mit *Tamino* können hochkomplexe XML-Files analysiert und abgefragt werden. Es ist portiert für alle gängigen Betriebssysteme.

Nachteile Für nicht-strukturierte Daten, Stored Procedures und Trigger-Mechanismen müssen eigene Server-Extensions selbst programmiert werden. Die Komplexität erfordert einen hohen Lernaufwand und einen Administrator, um das System optimal zu benutzen.

Kapitel 4

Systementwurf

4.1 Entwicklungsphasen

Wenn es um die Entwicklung komplexer Anwendungen geht, kommen sehr oft Verfahren aus dem Software Engineering zum Einsatz. Das bekannteste und wahrscheinlich am häufigsten angewandte Modell, das eine kontrollierte und koordinierte Softwareentwicklung ermöglicht, ist das Wasserfallmodell, das bereits von Winston W. Royce beschrieben wurde [Roy70]. Das Modell wurde weiterentwickelt von Boehm [Spi88].

Weiterhin kann man unterscheiden, ob es sich um einen herkömmlichen Entwicklungsprozess handelt (wie die Wasserfallmethode beschreibt) oder um einen so genannten komponentenbasierten Softwareentwicklungsprozess. Bei der Softwareentwicklung mit Komponenten werden zum Bau von Softwaresystemen zuvor entwickelte wiederverwendbare Bausteine genutzt.

Bei der Entwicklung des Softwaresystems ASIX (Automatic Search in XML) war es sinnvoll, das Wasserfallmodell als iterative Methode mit dem komponentenbasierten Entwicklungsprozess zu kombinieren, wie es in Abbildung 4.1 auf der folgenden Seite dargestellt ist.

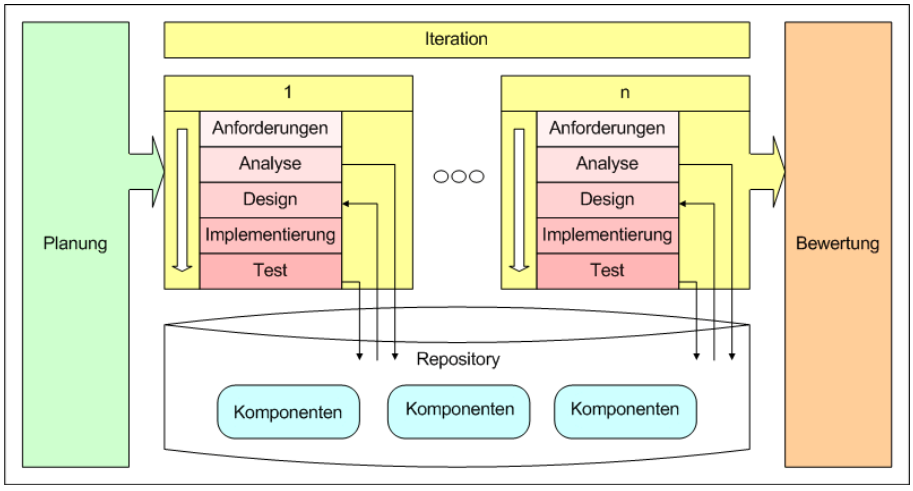


Abbildung 4.1: Komponentenbasierter iterativer Entwicklungsprozess

4.2 Anforderungen und Analyse

4.2.1 Ist-Analyse

Für die Suche in leichtgewichtigen XML-Datenbanken besteht bislang die Möglichkeit, vorgefertigte Module, die eine Anfragesprache realisieren, einzusetzen. Diese Module werden pro Anfrage auf die leichtgewichtige XML-Datenbank gestartet und liefern das Ergebnis zurück. Jede weitere gleichlautende Anfrage durchsucht wie die erste die gesamte XML-Datenbank. Daher sind die Antwortzeiten gleichlang (Formel 4.2.1).

Formel 4.2.1

$$\begin{aligned} Antwortzeit_1(Anfrage_1) &= \ldots = Antwortzeit_n(Anfrage_1) \\ &\vdots \\ Antwortzeit_1(Anfrage_m) &= \ldots = Antwortzeit_n(Anfrage_m) \end{aligned}$$

$m, n \in N$

4.2.2 Ziele und Anforderungsanalyse

Das beschriebene Verfahren zur Anfrage in leichtgewichtigen XML-Datenbanken kann dann nicht befriedigen, wenn – wie oft der Fall – Anfragen wiederholt gestellt werden. Dann wird die Redundanz bei der wiederholten Durchforstung des gesamten Datenbestandes offensichtlich.

Ein neu zu entwickelndes System, das die Historie bereits gestellter Anfragen berücksichtigt, könnte die Performance bei Anfragen steigern. Dazu wäre ein "Gedächtnis" von Nutzen, in dem einmal gemachte Erfahrungen gesammelt und bei Bedarf wieder abgerufen werden können. Das setzt voraus, dass es möglich sein muss, dieses "Gedächtnis" schneller abzurufen, als eine neue Anfrage an das System zu stellen.

Erstrebenswert ist, dass bereits ab der zweiten gleichartigen Anfrage ein Zeitgewinn eintritt.

Es soll gelten:

Formel 4.2.2

$$Antwortzeit_2(Anfrage_k) > Antwortzeitneu_2(Anfrage_k)$$

$1 \leq k \leq m \in N$

Der Zeitgewinn ab der zweiten Anfrage kann dann, wie folgt, quantifiziert werden:

Formel 4.2.3

$$Antwortzeit_i(Anfrage_k) = Antwortzeitneu_i(Anfrage_k) + Zeitgewinn_i(Anfrage_k)$$

$1 \leq k \leq m \in N$

$2 \leq i \leq n \in N$

Bei wachsendem n erhöht sich der Zeitgewinn für alle gleichlautenden Anfragen. Der Zeitgewinn $Zgewinn(Anfrage_k, n)$ für die Anfrage k nach $n \geq 2$ Anfragen ist dann:

Formel 4.2.4

$$Zgewinn(Anfrage_k, n) = \sum_{i=2}^{n} Zeitgewinn_i(Anfrage_k)$$

$1 \leq k \leq m \in N$

$n \in N, n \geq 2$

Gleichzeitig zu berücksichtigen ist, dass der Performancegewinn nicht durch die dabei zusätzlich zu speichernden Informationen verloren geht. Es ist also neben des effektiveren Zugriffs auch die Speichereffizienz dieses Overheads entsprechend zu beachten.

Wünschenswert ist es auch, wenn die gesammelten "Erfahrungen" des Systems bei Änderungen in der Datenbank oder der Inhalte einzelner Dateien nicht verloren gehen. Insofern sollte es sich selbst reorganisieren können. Die erste Anfrage an ein solches System kann selbstverständlich auf keinerlei Information zurückgreifen. Mit jeder weiteren Anfrage wächst aber das "Gedächtnis" des Systems.

Reorganisation und Gedächtniswachstum des Systems müssten weitestgehend automatisch ablaufen. Insofern kann das System dann als selbstadaptierend bezeichnet werden.

4.3 Strategie der Realisierung

Zu entwickeln ist ein Verfahren, das Anfragen an eine leichtgewichtige XML-Datenbank zulässt und dabei berücksichtigt, ob bereits Ergebnisse vorangegangener Anfragen vorliegen. Ist das der Fall, muss keine Anfrage an die Datenbank gestellt werden, wenn sichergestellt ist, dass auf eine andere Art das Anfrageergebnis schneller zu erhalten ist.

Zunächst war das Problem zu lösen, die Ergebnisse bereits gestellter Abfragen in geeigneter Form für weitere Anfragen schnell abrufbar bereit zu halten. Dafür wurde eine Methode der Indexierung gewählt. Anders als sonst werden aber nicht die XML-Quellen selbst indiziert, sondern das aus einer XML-Anfrage gewonnene Ergebnis wird als Pfadindex in einer Indexdatei gespeichert. Um die Anforderungen an die Speichereffizienz zu wahren, dürfen Indextabellen aber nicht wahllos entstehen. In Abhängigkeit der vorangegangenen Suchanfragen werden nur dann Indexeinträge erzeugt, wenn die Forderung nach einer minimalen mittleren Suchzeit oder die Anzahl gleicher, in der Vergangenheit getätigter Suchanfragen, überschritten wurde.

Die Grundidee des Verfahrens besteht nun darin, bei jeder Anfrage zu ent-

scheiden, ob eine XML-Anfrage ausgelöst werden soll oder ob das Ergebnis der Anfrage bereits aus den Indexfiles gewonnen werden kann.

Für die Realisierung des Verfahrens war zunächst zu berücksichtigen, dass die Suche über die Indexfiles schneller realisiert werden kann, als eine XML-Anfrage zu starten. Die Suche über Indexfileeinträge musste aber der XML-Anfrage selbst entsprechen. Werden die Indextabellen im ASCII-Format aufgebaut, ist eine sehr schnelle Informationsgewinnung möglich. Dazu ist es aber unumgänglich, die XML-Anfrage in eine SQL-Anfrage zu transformieren. Wie schon gezeigt wurde, gibt es eine weitgehende syntaktische Ähnlichkeit zwischen XQL und SQL im Aufbau der Queries. Geschieht diese Transformation automatisch, bleibt sie für den Nutzer des Verfahrens transparent. Da dies komfortabel unter Verwendung Regulärer Ausdrücke geschehen kann, wurde für die Realisierung XQL als Anfragesprache und Perl wegen der Unterstützung Regulärer Ausdrücke als Programmiersprache gewählt.

Der Nutzer des Systems kann nun Suchanfragen immer in XQL stellen. Ob in der XML-Datenbank oder in den Indextabellen gesucht wird, entscheidet das System.

Für den Fall, dass Änderungen in der XML-Datenbank vorgenommen wurden, werden Funktionen benötigt, die den Index immer auf dem neuesten Stand halten.

Zur Umsetzung dieses Verfahrens wurde das System "Automatic Search in XML" (ASIX) entwickelt.

4.4 Das Schichtenmodell

Die vier ASIX-Schichten *Userlayer*, *Applicationlayer: Logic Components*, *Applicationlayer: Search Moduls* und *Datalayer* werden von einer unsichtbaren Hülle umgeben. Das ist zum einen *dnotify*, welches im Hintergrund läuft und deshalb vom Nutzer nicht unmittelbar wahrgenommen wird. Zum anderen die XML-Datenbank, die das System verwendet.Sie kann vom Nutzer durch das Hinzufügen, Löschen oder Ändern von Files verändert werden kann, was wiederum Auswirkungen auf das System hat. Im Moment einer Änderung ist der Index nicht mehr aktuell und muss deswegen aktualisiert werden.

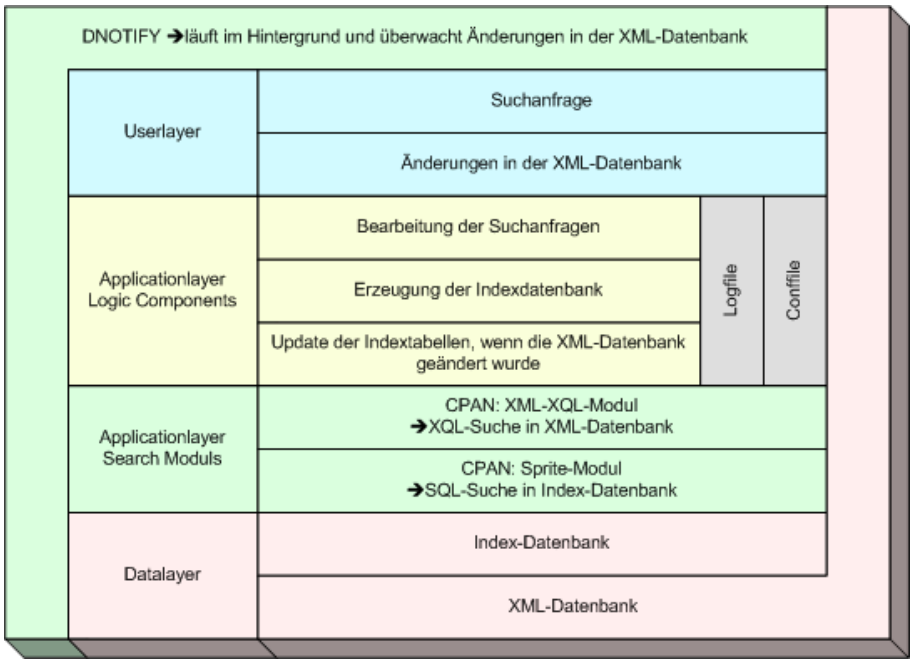


Abbildung 4.2: Das Schichtenmodell

Userlayer Die *Userlayer* ist die einzige Schicht, die der Nutzer sieht. Hier können Suchanfragen an das System gerichtet und die XML-Datenbank verändert werden. Der interne Vorgang bei einer Suchanfrage bleibt dem Anwender verborgen. Das System löst alle Aufgaben vollautomatisch und meldet sich erst mit dem Ergebnis der Anfrage wieder zurück. Ob in den XML-Files oder im Index gesucht wurde, ist für den Nutzer transparent. Ändert der Nutzer die XML-Datenbank, wird durch das im Hintergrund laufende *dnotify* der Aktualisierungsprozess angestoßen. Das Update läuft auch im Hintergrund des Betriebssystems und ist deswegen für den Nutzer nicht sichtbar.

Applicationlayer: Logic Components In dieser Schicht befindet sich die Logik für die Bearbeitung von Anfragen und das Management der Indextabellen. Die Programmodule sind abhängig von den Parametern des Konfigurationsfiles */etc/asix.conf*, die ein Administrator verändern kann. Im diesem File sind die Pfade der Bibliotheken und des Indexes, der Name des Logfiles, sowie Parameter zur Indexerstellung gespeichert. Im Logfile */var/ASIX/logfile.lg* wird jede Suchanfrage, die mit XML-XQL ausgeführt wurde, protokolliert. Später werden diese Daten zusammen mit den Parametern aus dem Konfigurationsfile verwendet, um den

Index aufzubauen bzw. zu erweitern. Durch *dnotify* werden die Updateroutinen, die den Index aktualisieren, aufgerufen. Alle Module verwenden die Komponenten der *Applicationlayer: Search Moduls* für die Suchanfragen. Wird während einer Suchanfrage festgestellt, dass bereits ein Index existiert, erfolgt die Suche in den Indexfiles. Die Indexfiles werden so aufbereitet, dass nach vorheriger Transformation der XQL-Query nach SQL die Suche mit dem schnellen Sprite-Modul[1] ausgeführt werden kann.

Applicationlayer: Search Moduls Das XML-XQL-Modul aus dem Comprehensive Perl Archive Network (CPAN) wird immer dann aufgerufen, wenn eine XQL-Query auf den XML-Files ausgeführt werden soll. Dazu wird die Query zusammen mit den Namen der XML-Files an das Modul übergeben. Das Ergebnis einer Anfrage wird anschließend zur Ausgabe auf dem Bildschirm aufbereitet. Wurde die Suche aus dem Suchmodul gestartet, werden gleichzeitig die Suchzeit und das Tagesdatum der Suche zusammen mit den Pfaden der Query im Logfile gespeichert.

Das Sprite-Modul wird immer dann aufgerufen, wenn das Suchmodul festgestellt hat, dass für eine Suchanfrage bereits ein Indexfile existiert.

Datalayer Die *Datalayer* besteht aus der XML-Datenbank, die vom Nutzer verändert werden kann, sowie aus der Indexdatenbank, auf die der Nutzer keinen direkten Zugriff hat. Die Indexdatenbank steht in einem Verzeichnis, das im Konfigurationsfile deklariert wurde und nur von einem Nutzer mit Superuserrechten verändert werden darf[2].

4.5 Das Komponentenmodell

Der komponentenbasierte Aufbau des Systems ASIX richtet sich nach den einzelnen Funktionen der Bausteine. Mit diesem Modell läßt sich die Funktion jedes Bausteins abgrenzen und genau beschreiben. *dnotify* spielt eine Sonderrolle, weil es zum einen kein Perlprogramm ist und zum anderen nur indirekt mit den Funk-

[1] siehe Kapitel 5.1.1 auf Seite 65

[2] Soweit das verwendete Betriebssystem die Vergabe von Rechten unterstützt.

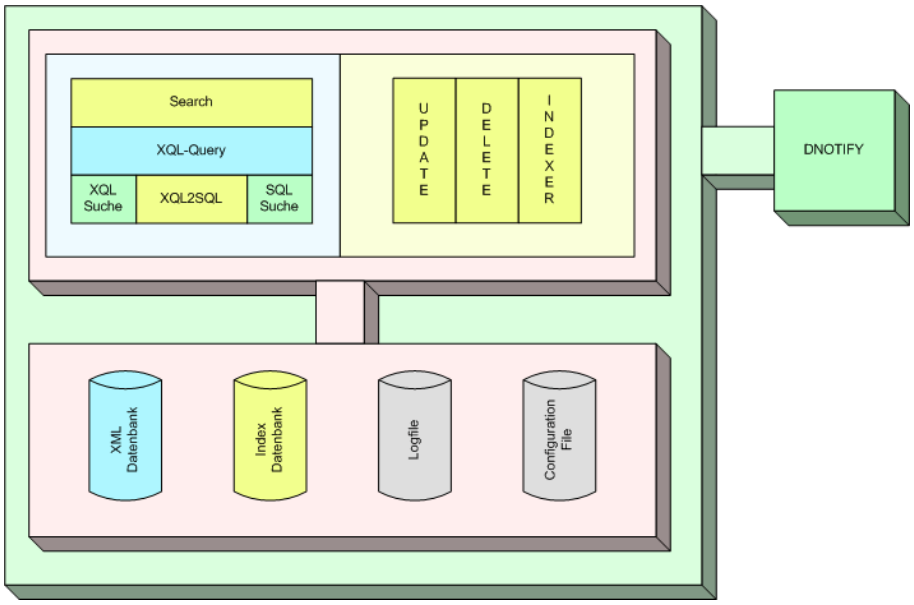


Abbildung 4.3: Das Komponentenmodell

tionen des Systems verbunden ist.

dnotify umschließt alle Komponenten und es läuft unsichtbar im Hintergrund des Systems. Einmal vom Administrator gestartet, bleibt es während der gesamten Laufzeit dem Nutzer verborgen. *dnotify* überprüft die Verzeichnisse, in denen die XML-Files liegen, auf Änderungen. Bemerkt es eine Änderung, wird sofort eine adäquate Befehlsfolge ausgeführt. Wird ein XML-Dokument geändert oder eines hinzugefügt, startet *dnotify* die Updatefunktionen von ASIX. Wird ein XML-Dokument gelöscht, startet es entsprechend die Deletefunktion von ASIX.

Search Werden Anfragen an das System gestellt, so ist das Searchmodul für deren Lösung zuständig. Suchanfragen werden immer in Form einer XQL-Query übergeben. Das Suchmodul muss überprüfen, ob für diese Query bereits ein Indexfile existiert. Dazu muss es die Pfade aus der XQL-Query extrahieren und daraus einen Indexnamen erzeugen. Indexfilenamen bestehen aus einem einfachen Pfadausdruck gemäß Definition 3.5.1 auf Seite 40, also aus den Pfaden ihres Inhalts, die mit Punkten voneinander getrennt sind und der angehängten Dateiendung `idx`.

```
pfad1.pfad2.pfad3.idx
```

Stellt das Programm fest, dass ein Index existiert, wird die XQL-Query nach SQL transformiert. Anschließend wird die Suchanfrage mit Sprite ausgeführt. Existiert noch kein Indexfile, wird die XQL-Query an XML-XQL weitergeleitet. In diesem Fall werden die Pfade der Query zusammen mit der Suchzeit und dem aktuellen Jahrestag im Logfile */var/ASIX/logfile.lg* protokolliert und später zur Erstellung des Indexes verwendet.

Update Die Updatefunktion tritt immer dann in Kraft, wenn Änderungen in den XML-Files festgestellt wurden. Das Feststellen von Änderungen erledigt das externe Programm *dnotify*. Werden neue XML-Dokumente der Datenbank hinzugefügt oder existierende verändert, erfordert das eine andere Behandlung als das Löschen von Dateien. Das Updatemodul ermittelt die Files, die von der Änderung betroffen sind. Anschließend werden alle Indextabellen durchsucht und die entsprechenden Einträge aktualisiert. D.h., nicht der gesamte Index wird neu erzeugt, nur Indextabellen, in denen Einträge geänderter XML-Dokumente stehen, werden aktualisiert.

Delete Das Löschen geschieht ähnlich wie das Update. Hierzu muss ein zweites *dnotify* laufen, welches das Löschen überwacht. Die Deletefunktion stellt fest, welche Datei gelöscht wurde. Anschließend müssen alle Indexfiles nach Einträgen der gelöschen Datei durchsucht werden. Wenn diese Einträge aus den Indexfiles entfernt wurden, ist der Index wieder aktuell.

Indexer Der Indexer wird gestartet, wenn Indexfiles erzeugt werden sollen. Dies kann zum Beispiel immer nachts passieren. Die gewonnenen Daten aus der Logdatei werden mit den Parametern der Konfigurationsdatei verglichen. Wird festgestellt, dass ein oder mehrere Indexfiles erzeugt werden müssen, werden entsprechende XQL-Queries ausgestellt. Das Ergebnis der XQL-Suche wird in Tabellenform gebracht und als Indexfile abgespeichert.

Daten In der XML-Datenbank befinden sich die Original-XML-Dokumente. Die Indexdatenbank kann als kleiner Auszug der XML-Datenbank aufgefasst werden. Hier werden die Teile der XML-Dokumente, nach denen häufig gesucht

wird, gespeichert. Im Logfile werden alle ausgeführten XQL-Anfragen protokolliert. Parameter, wie Verzeichnisse und Kriterien zur Indexerstellung, werden in dem Konfigurationsfile aufbewahrt.

4.6 Perl

4.6.1 Überblick

Perl ist eine sehr mächtige, interpretierende Programmiersprache. Perlprogramme werden deshalb auch Perlscripte genannt. Sie vereint die Vorteile aus C, awk, sh, sed und weiteren UNIX-Tools. Es gibt zwei typische Formen von Programmiersprachen. Der Quelltext von Scriptsprachen wird durch einen Interpreter zeilenweise ausgeführt. Bei rein kompilierenden Sprachen wird zuerst der Quelltext kompiliert und der entstandene Binärcode wird vom Betriebssystem ausgeführt. Perl kombiniert beide Formen. Der Code wird zur Laufzeit in eine Zwischenform interpretiert, die dann sehr schnell kompiliert und ausgeführt wird. Mit diesem Verfahren sind Perl-Programme fast ebenso schnell wie C-Programme.

4.6.2 Reguläre Ausdrücke

Eine herausragende Stärke von Perl ist die Implementierung Regulärer Ausdrücke. Reguläre Ausdrücke wurden nicht in Perl neu erfunden, sondern gehören in die Theorie Formaler Sprachen. Dort werden sie wie folgt definiert:

Definition 4.6.1 *(Regulärer Ausdruck) Sei $\sum$ ein Alphabet. Ein Regulärer Ausdruck ist eine Zeichenfolge, die aus Elementen von $\sum$ und einigen weiteren Zeichen (Klammern, ., —, *, ε) nach folgenden Regeln rekursiv gebildet wird:*

1. *Jedes Zeichen $a \in \sum$ ist ein Regulärer Ausdruck.*

2. *Ein spezielles Zeichen ε für das leere Wort ist ein Regulärer Ausdruck.*

3. *Sind R und S Regulärer Ausdrücke, so auch $R \cdot S$. Hierdurch wird die Konkatenation ausgedrückt.*

4. *Sind R und S Regulärer Ausdrücke, so auch $R \mid S$. Hierdurch wird die Vereinigung ausgedrückt.*

5. *Ist R ein Regulärer Ausdruck, so auch R^*. "*" bedeutet n-malige Wiederholung. $n \geq 0$ beliebig.*

6. *Genau die durch die Schritte 1 - 5 konstruierbaren Ausdrücke sind Regulär.*

In der Welt der Programmiersprachen werden Reguläre Ausdrücke, auch Regular Expressions genannt, so definiert:

Definition 4.6.2 *Ein Regulärer Ausdruck ist ein Suchmuster, um übereinstimmende Muster in einer Eingabe zu finden.*

Definition 4.6.3 *Ein Regulärer Ausdruck ist ein Muster, das mit einer Zeichenkette verglichen wird.*
Aus [SP02]

Somit kann eine Zeichenkette überprüft werden, ob sie einem bestimmten Muster entspricht. Es ist auch möglich, Teile eines Strings, die auf ein Muster passen, durch einen anderen String zu ersetzen. Perl ist eine semantische Obermenge aller UNIX-Programme, die Regular Expressions verwenden (grep, sed, awk, etc.). Alle Regulären Ausdrücke, die mit einem dieser Programme gebildet werden können, sind auch in Perl ausdrückbar. Bei Regulären Ausdrücken haben verschiedene Sonderzeichen eine besondere Bedeutung, die je nach verwendetem Programm unterschiedlich sein kann.

Muster Jeder Regulärer Ausdruck ist ein Muster. Die Teile eines Musters können auf einzelne Zeichen eines Strings oder auf einen Teilstring passen. Damit Perl erkennt, dass es sich um ein Muster handelt, werden sie immer in ein Slashpaar eingeschlossen (/ ... /). Es ist möglich nach einem beliebigen Zeichen (/./), einem bestimmten Zeichen (/a/) und nach einer Zeichenklasse zu suchen (/[a-z]/). Mit dem ^-Zeichen können Zeichenklassen negiert werden (/[^a-z]/). Bestimmte

Zeichenklassen lassen sich auch durch vordefinierte Konstrukte ausdrücken (siehe Tabelle 4.1[3])

Konstrukt	**Klasse**	**neg. Konstrukt**	**neg. Klasse**
eine Ziffer: \d	[0-9]	\D	[^0-9]
ein Wortzeichen: \w	[a-zA-Z0-9_]	\W	[^a-zA-Z0-9_]
Leerraum: \s	[\f\n\r\t]	\S	[^\f\n\r\t]

Tabelle 4.1: Abkürzungen von Zeichenklassen

In einer Regular Expression können Multiplikatoren eingesetzt werden, die die Anzahl des Auftretens des vorherigen Zeichens oder der Zeichenklasse bestimmen. Der Asterisk (*) steht für null bis unendlich, d.h., das Zeichen, welches vor dem Asterisk steht, kann niemals, einmal oder beliebig oft hintereinander auftauchen. Mit dem Plus-Zeichen (+) kann angegeben werden, dass ein Zeichen mindestens einmal auftreten muss. Das Fragezeichen (?) bezeichnet, dass ein Zeichen null-mal oder einmal auftreten darf. Mit geschweiften Klammern lassen sich beliebige Multiplikatoren bilden, z.B. /3{0,5}/, die 3 darf höchstens 5-mal nacheinander vorkommen oder /7{7,}/, die 7 muss mindestens 7-mal hintereinander auftreten.

Bei langen, komplexen Regular Expressions kann es oft zu Wiederholungen von Teilmustern kommen. Diese können, wenn sie jeweils das erste Mal auftreten, in runden Klammern gespeichert werden und anschließend mit einem Backslash gefolgt, von einer Zahl, die die Nummer des Klammerpaares von links gerechnet angibt, wiederverwendet werden. Das Muster /reg(.*\s)Exp\1/ trifft auf alle Zeichenketten zu, die die Zeilzeichenkette *reg*, gefolgt von *beliebig vielen Zeichen, außer Newline*, gefolgt von einem *Leerzeichen*, gefolgt von *Exp*, wieder *beliebig vielen Zeichen, außer Newline* und gefolgt von einem *Leerzeichen* enthält.

Regex-Maschine Die Funktionsweise von Regex-Maschinen basiert auf zwei wesentlichen Regeln. Die erste lautet: *"Der früheste Treffer gewinnt."* oder *"Der Treffer gewinnt, der am frühesten beginnt."*[4] Verglichen werden Strings immer von links, d.h., für das erste Zeichen, auf das eine Regel paßt, gilt sie als *wahr*. An

[3] Eine ausführliche Liste definierbarer Konstrukte befindet sich im Anhang B.3 auf Seite 115

[4] aus [Fri98] Seite 95

jeder Stelle im String werden alle möglichen Kombinationen der Regex versucht, um einen Treffer zu landen. Die erste Stelle, an der gesucht wird, ist vor dem ersten Zeichen. Nachdem alle Permutationen angewandt wurden und es noch keinen Treffer gab, wird vor das zweite Zeichen gesprungen. Dort beginnt der Test wieder von vorn. Bis ein Treffer gefunden wurde, werden so alle Stellen im String sequentiell besucht. Schlagen alle Versuche fehl, gilt die Regex als *falsch*.

Beispiel 4.6.1 *Die Regex* `/ist/` *meldet im String*

```
Der Benzinkanister ist im Kofferraum
```

somit keinen Treffer bei `ist`*, sondern schon bei* `Benzinkanister`*.*

Dieser Effekt spielt auch bei der Verwendung der Alternative eine Rolle. So wird die angegebene Reihefolge der Alternativen nicht berücksicht. Gibt es einen Treffer auf einen Wert, gilt die Regex als gelöst.

Beispiel 4.6.2 *Die Regex* `/(Koffer|Benzin)/` *meldet im String*

```
Der Benzinkanister ist im Kofferraum
```

einen Treffer bei `Benzin`*.*

Die zweite Regel besagt, dass manche Metazeichen[5] (Quantoren) gierig sind, also: "Teile von Regulären Ausdrücken, die in variabler Anzahl vorkommen dürfen, versuchen immer, auf so viele Zeichen wie nur möglich zu passen. *Sie sind gierig.*"[6] D.h., eine Regex trifft immer auf die Maximalzahl von Zeichen zu, die auf sie treffen und für die die gesamte restliche Regex gültig ist. So würde das `\d+` von `/\d+7/` auf die gesamte Zahl `1234567` passen. Da aber gefordert ist, dass nach mindestens einer Ziffer eine Sieben folgt, muss sich `\d+` mit `123456` begnügen. [Fri98]

4.6.3 CPAN

Das *Comprehensive Perl Archive Network* (CPAN[7]) ist ein FTP-Server mit einer großen Sammlung von mehr als 4000 getesteten und frei verfügbaren, formaliserten Perlsoftwareprodukten und Dokumentationen. Dieses Repository ist in

[5] ?, *, +, {min,max}

[6] aus [Fri98] Seite 98

[7] Angelehnt an CTAN (Comprehensive TEXArchive Network)

seiner Form einzigartig und weltweit anerkannt. Die Sammlung enthält auch ein Perlmodul gleichen Namens CPAN. Es regelt den Download und das Installieren von Modulen aus dem CPAN-Archiv. Um ständige Verfügbarkeit zu gewährleisten gibt es weltweit über 175 Mirrors, die ständig gepflegt werden. Nach einem Upload eines neuen Modules ist es innerhalb von Stunden in allen Mirrors verfügbar, [cpa03].

4.6.4 POD

Plain Old Documentation (POD) ist ein Dokumentationsformat für Perl-Code. Die Dokumentation muss im ASCII-Format erfolgen. Dabei kann es sich um Hinweise zu bestimmten Funktionen des Programms oder auch um Informationen über den Entwickler handeln. Sie kann sich an jeder Stelle im Quelltext befinden und besteht aus Formatierungsanweisungen. Der Vorteil besteht darin, dass sich bereits im Quelltext selbst die Dokumentation dazu befindet. Das Format erinnert an Markierungssprachen wie HTML oder LaTeX. Mit den Anweisungen erfolgt eine Strukturierung nach Köpfen, Tabellen, Listen und anderen Sektionen. Mit Formatierungsanweisungen kann bestimmer Text fett oder kursiv, etc. hervorgehoben werden[8]. Mit Konvertierungsprogrammen[9] läßt sich der POD-Teil eines Dokuments in LaTeX, HTML, Manpage, etc. wandeln, [Haj98].

4.6.5 Vor- und Nachteile von Perl

Ein großer Vorteil für den Programmentwickler ist die Verfügbarkeit einer Programmiersprache. Perl gibt es kostenlos im Internet und mit dem CPAN-Archiv steht eine große Anzahl von getesteten Modulen zur freien Verfügung. Aus diesem Grund hat sich eine sehr große Anwendergemeinschaft gebildet mit unzähligen Portalen und Foren zum Programmieren mit Perl. Ein weiterer wichtiger Punkt ist die Fähigkeit von Perlprogrammen auf verschiedenen Plattformen zu laufen. Dazu gehören alle gängigen UNIX-Versionen, Macintosh und Microsoft Windows. Mit Perl kann man sowohl kleine, sehr effektive, als auch umfangreiche, komple-

[8] Eine ausführliche Liste definierbarer Konstrukte befindet sich im Anhang in den Tabellen B.7 auf Seite 118 und B.8 auf Seite 118

[9] zum Beispiel pod2html

xe Programme schreiben. Dadurch, dass der Code vor dem Ausführen kompiliert wird, werden Fehler schon vor der Laufzeit erkannt. Funktionen können in andere Programmiersprachen eingebettet werden und umgekehrt. So kann die Ausführung kritischer Funktionen beschleunigt werden, in dem diese zum Beispiel in C implementiert und dann in Perl eingebunden werden. Perl unterstützt das imperative und das objektorientierte Programmierparadigma.

Ein Perlscript wird vor jedem Ausführen interpretiert. Wie bei allen Scriptsprachen ist der Quellcode offengelegt und damit für alle sichtbar. Da Perlprogramme nicht vom Betriebssystem ausgeführt werden, muss der Interpreter auf der Zielmaschine installiert sein. Der Code von Perl kann unter Umständen sehr kryptisch und schwer lesbar sein. [Haj98]

4.6.6 Kriterien für die Auswahl von Perl

Perl wurde als Programmiersprache vor allem aus zwei Gründen gewählt:

1. Mit den Regulären Ausdrücken bietet Perl ein mächtiges Werkzeug, mit dessen Hilfe die Transformation der XQL-Anfragen nach SQL ressourcenschonend möglich gemacht werden kann.

2. Der Reichtum an Funktionen und Modulen im CPAN Repository bietet die Möglichkeit, den Softwareentwicklungsprozeß komponentenbasiert iterativ zu gestalten.

Kapitel 5

ASIX

5.1 Bausteine des Systems

ASIX ist ein System, das aus vielen kleinen Bausteinen besteht, die alle von einannder abhängig sind. Jeder Baustein erfüllt eine spezielle Aufgabe, die von einem anderen verwendet wird. Aus diesem Grund ist ASIX sehr flexibel und kann mit wenigen Veränderungen einer neuen Aufgabe angepaßt werden, sofern dies notwendig ist. Durch das verwendete Schichtenmodell bleiben dem Nutzer systeminterne Vorgänge verborgen. Im Komponentenmodell sind die einzelnen Module und deren Funktion klar erkennbar.

5.1.1 Sprite

Sprite ist ein Perl-Modul, das von Shishir Gundavaram[1] entwickelt wurde und im CPAN verfügbar ist. Mit SQL-Anweisungen können ASCII-Tabellen, deren Spalten mit Tabstop getrennt sind, durchsucht werden. Sprite zeichnet sich vor allem durch seine hohe Performance aus. So sind selbst bei vielen und großen Tabellen die Antwortzeiten sehr gering. Weiterhin unterstützt Sprite Regular Expressions.

5.1.2 XML-XQL

Dieses Perl-Modul verwendet für die Suche das Document Object Model (DOM), d.h., bevor die Werte der Elemente in einem XML-File mit der Anfrage verglichen werden können, muss zuerst ein Baum der Struktur des Files im Hauptspeicher

[1] E-Mail: shishir@ora.com

des Systems erstellt werden. In einem vorangegangenen Benchmark wurde untersucht, von welchen Parametern die Suchzeit in XML-Files abhängt. Es wurden Testsuchen mit verschieden strukturierten XML-Files durchgeführt. Dabei hat sich gezeigt, dass mit zunehmender Anzahl von XML-Files die Antwortzeiten exponentiell zunehmen - bei gleicher Knotenanzahl und gleicher Strukturtiefe. Weil bei jeder Suche alle Files durchsucht werden, sind Position und Anzahl von wahren Ergebnisses für das Suchzeitverhalten nicht ausschlaggebend, d.h., eine Anfrage mit Treffer im ersten Dokument dauert genauso lange wie eine mit Treffern in allen Dokumenten und eine mit einem Treffer im letzten Dokument. Im Benchmark hat sich auch gezeigt, dass das Vorkommen von Bedingungen in einer Anfrage keinen Einfluß auf die Antwortzeiten hat (siehe auch Anhang A auf Seite 103).

5.1.3 dnotify

dnotify ist ein Programm, welches immer dann ein angegebenes Kommando ausführt, wenn sich der Inhalt des überwachten Verzeichnisses ändert. Es wird in der Kommandozeile gestartet und kann zwei verschiedene Argumente aufnehmen: ein oder mehrere Verzeichnisse, die überwacht werden und das Kommando, das ausgeführt wird, wenn eine Änderung in einem der übergebenen Verzeichnisse festgestellt wurde. Die wichtigsten Optionen für das Auslösen werden in Tabelle 5.1 auf der nächsten Seite zusammengefaßt.

dnotify wurde in C geschrieben und benutzt das "linux kernel directory notification feature" zur Verzeichnisüberwachung. Aus diesem Grund fragt es nicht ständig die Files ab, sondern "schläft" bis ein Event auftritt. Allerdings kann aufgrund von Einschränkungen im Kernelinterface nicht das betroffene File ausgegeben werden, sondern muss durch externe Funktionen bestimmt werden.

5.1.4 ASIX-Modul

Das ASIX-Modul hat alle Funktionen eines Hauptprogramms. Von diesem Modul aus werden alle Funktionen, durch Funktionsaufrufe gesteuert. Alle Funktionen sind durch den Aufruf dieses Moduls erreichbar durch die Angabe eines bestimmten Parameters. Die einzelnen Module erfüllen separate Aufgaben, die zur Su-

Option		**Bedeutung**
Events, bei denen das Komanndo ausgelöst wird:		
-A	–access	ein File wurde aufgerufen
-M	–modify	ein File wurde verändert
-C	–create	ein File wurde neu erzeugt
-D	–delete	ein File wurde unlinked
-R	–rename	ein File wurde renamed
-B	–attrib	nach den Kommandos chmod und chown
-a	–all	entspricht allen Optionen
Allgemeine Optionen:		
-e	–execute= COMMAND...	Kommando, das ausgeführt wird, wenn einer der oben genannten Fälle eintritt.
-r	–recursive	Unterverzeichnisse rekursiv überwachen

Tabelle 5.1: dnotify-Optionen

che, Transformation der XQL-Query, sowie zur Verwaltung der Indexdatenbank benötigt werden. Abbildung 5.1 auf der folgenden Seite zeigt das Zusammenspiel der einzelnen Module von ASIX.

Im Folgenden wird davon ausgegangen, dass ASIX als Stand-Alone-Anwendung betrieben wird. Ist ASIX in ein übergeordnetes System integriert, sind die Kommandozeilenanweisungen als Übergabeparameter innerhalb eines systeminternen Funktionsaufrufes zu sehen. Der Aufruf von ASIX in der Kommandozeile hat immer folgende Form:

```
$ asix.pl Funktionsparameter Datenparameter Datenparameter ...
```

Über den **Funktionsparameter** wird gesteuert, welche Operation ausgeführt werden soll. Bei Angabe von `asix -s` wird das Modul mit den Suchfunktionen zur Suche in einer XML-Datensammlung aufgerufen (siehe Kapitel 5.2 auf Seite 69). Zur Erstellung neuer Indexfiles muss der Parameter `-i` verwendet werden (siehe Kapitel 5.3 auf Seite 75). Für das Update-Modul existieren zwei Parameter. Soll eine XML-Datei gelöscht werden, so kann dies nicht direkt erfolgen, sondern muss auch über ASIX laufen. Denn nur so kann gewährleistet werden, dass der Index

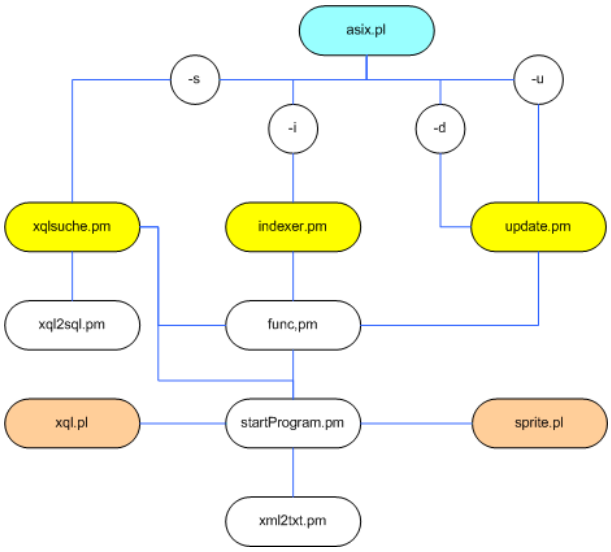


Abbildung 5.1: Modulstruktur von ASIX

immer aktuell ist. Der Aufruf dieser Funktion erfolgt mit `asix -d`. Das Programm *dnotify* läuft im Hintergrund und überprüft ständig, ob sich im Verzeichnis der XML-Files etwas geändert hat. Wurde ein XML-File geändert oder hinzugefügt, ruft *dnotify* ASIX mit der Option `-u` auf (siehe Kapitel 5.4 auf Seite 79).

Die **Datenparameter** sind je nach Funktion unterschiedlich, deshalb wird darauf in den jeweiligen Kapiteln eingegangen. Alle Funktionen sind abhängig von der Konfigurationsdatei */etc/asix.conf*[2]. Unmittelbar nach dem Programmstart wird sie eingelesen und es werden globale Environmentvariablen der zu verwendenden Pfade, der Name der Logdatei sowie die Parameter $MAX_LASTACCESS, $MIN_QUERYCALLS und $MAX_SEARCHTIME gesetzt.

Die Variable $MAX_LASTACCESS wird verwendet, um den Index möglichst klein zu halten. Da bei der Suche das Tagesdatum mitgespeichert wird, können so Indexfiles für Suchanfragen, die länger als x Tage zurückliegen, wieder gelöscht werden. Die Variablen $MIN_QUERYCALLS und $MAX_SEARCHTIME regeln die Erzeugung von Indextabellen. Übersteigt die Anzahl einer Anfrage $MIN_QUERYCALLS oder beträgt die Suchdauer einer Query mehr als $MAX_SEARCHTIME Sekunden, wird ein Index erzeugt.

[2]Das Listing der Datei und die vollständige Parameterliste befinden sich in Anhang C.1 auf Seite 119

5.2 Die Anfrage

Übergabe der Query Ausgehend von der Annahme, dass ASIX auf Kommandozeilenebene ausgeführt wird, muss die Angabe der Parameter für die Suche wie folgt lauten:

```
$ asix.pl -s "XQL-Query" "XML-Files"
$ asix.pl -s "/group/assistant/name/lastname
             [/group/assistant/private/age < 25]" "*.xml"
```

Auf den Funktionsparameter folgen die Datenparameter "XQL-Query" und "XML-Files". Dem Programm werden diese Parameter als Kommandozeilenparameter in dem Array `@ARGV` übergeben, wobei das erste Arrayelement auch der erste Parameter ist.

Indextest Wie in dem Listing C.2 auf Seite 120 einer Indextabelle im Anhang zu erkennen ist, besteht der Dateiname aus den Pfaden, die im Tabellenkopf stehen. Das sind dieselben, die auch in der XQL-Query verwendet wurden. Anhand dieser Eigenschaft kann ein Programm die Extistenz einer Indextabelle prüfen ohne diese zu öffnen, da bereits am Dateiname der Inhalt erkannt werden kann. Der Dateiname der Indextabelle für die obige Query lautet wie folgt:

```
group__assistant__name__lastname.group__assistant__private__age.idx
```

Dies ist entscheidend für die Performance der Suchroutine. Nachdem die XQL-Query an das Programm übergeben wurde, muss getestet werden, ob zu dieser Query schon ein Index existiert bzw. zu den Pfaden, die in der Query verwendet wurden. Da die Namen der Indextabellen gleich den Pfaden in der XQL-Query sind, müssen die Pfade aus der Query extrahiert werden. Das geschieht mit einem Algorithmus dessen Herzstück die Regular Expressions sind. Der Algorithmus enthält folgende Teilschritte, die in Listing 5.1 auf der folgenden Seite nachvollzogen werden können:

1. Mit der Splitfunktion wird eine XQL-Query beim Auftreten eines Leerzeichens gesplittet. Alle Teile werden in dem Array *words* gespeichert (Zeile 50).

2. Alle Elemente von *words* werden auf das Enthaltensein eines Slashes (dann ist es ein Pfad oder eine Regular Expression) hin überprüft. Geht dem Slash kein Hochkomma (Quoting einer Regular Expression) vorher, wird das Element im Array *paths* gespeichert (Zeile 54-56).

3. Die ersten beiden Pfade müssen von der aufgehenden eckigen Klammer getrennt werden und als einzelne Elemente im Array *paths* gespeichert werden (Zeile 58-67).

4. Das erste Zeichen jeden Elements von *paths* wird entfernt, falls es nicht ein Slash ist (Zeile 70).

5. Doppelte Pfade werden durch die *uniq*-Funktions aus func.pm gelöscht (Zeile 79).

Listing 5.1: Auszug aus xqlsuche.pm

```
50 my (@words) = split ( /\s/, $ENV{XQL_QUERY} );
51
52 my @paths;
53
54 for (@words) {
55     push ( @paths, $_ ) if ( ( $_ =~ /\// ) and !( $_ =~ /'/␣)␣);
56 }
57
58 my $firstel = shift (@paths);
59 my @el      = split ( /\[/, $firstel );
60
61 if ( $paths[0] =~ /\.$/ ) {
62     unshift ( @paths, $el[0] );
63 }
64 else {
65     unshift ( @paths, $el[1] );
66     unshift ( @paths, $el[0] );
67 }
68
69 for ( my $i = 0 ; $i < scalar @paths ; $i++ ) {
70     substr( $paths[$i], 0, 1 ) = "" unless ( $paths[$i] =~ /^\// );
71 }
72
73 # doppelte Einträge zählen und speichern
74 my @pseudo;
75
```

```
76 my $pseudoref = \@pseudo;
77 my $pathsref  = \@paths;
78
79 ( $pathsref, $pseudoref ) = uniq( \@paths, \@pseudo );
```

Nachdem die Pfade extrahiert wurden, werden sie in der Funktion *build_index_file_name* in func.pm zum Indexnamen zusammengefügt. Ein Dateiname kann keine Slashes enthalten, weil der Slash Verzeichnisnamen trennt. Deshalb werden die Slashzeichen in den Pfaden durch doppelte Underlinezeichen (__) ersetzt. Ein einfaches Underlinezeichen ist nicht ausreichend, weil XML-Tagnamen auch das Underlinezeichen enthalten können. Die Ersetzung erfolgt mit dem Matchoperator und einer Regular-Substitution-Expression, siehe Listing 5.2 Zeile 83.

Listing 5.2: Auszug aus func.pm

```
69 sub build_index_file_name {
70
71     # @_: bedingungen
72
73     #      print "\n*************************************************";
74     #      print "\nStarting build_index_file_name...";
75
76     my @bed             = @_;
77     my $index_file_name = "";
78     foreach (@bed) { $index_file_name .= $_ . "."; }
79
80     $index_file_name .= "idx";
81     print "\n94 $index_file_name";
82
83     $index_file_name =~ s/\//__/g;
84
85     # ersten _ löschen
86     substr( $index_file_name, 0, 1 ) = "" if ( $index_file_name =~ /^_/ );
87
88     return $index_file_name;
89 }
```

Die Funktion *exist_index_file* überprüft mit Hilfe des Befehls `-e` (Zeile 150 in Listing 5.3) in einer `IF...THEN...ELSE`-Bedingung, ob eine bestimmte Datei in einem Verzeichnis existiert. Ein Öffnen der Datei ist nicht notwendig, da am Dateinamen bereits der Inhalt erkannt wird.

Listing 5.3: Auszug aus func.pm

```
144 sub exist_index_file {
145
146     # @_[0]: dateiname ohne verzeichnisprefix
147
148     my $dateiname = $ENV{VAR_PATH} . $_[0];
149     print "\nExistiert␣$dateiname???";
150     if ( -e $dateiname ) {
151         print "\nIndexfile␣existiert!";
152
153         # -> Suche mit Sprite
154         return (1);
155     }
156     else {
157         print "\nIndexfile␣ex.␣nicht!";
158
159         # -> Indexfile anlegen
160         return (0);
161     }
162 }
```

Wenn festgestellt wird, dass es zu dieser Query schon ein Indexfile gibt, wird die XQL-Query nach SQL transformiert. Existiert noch kein Indexfile, wird die Query unverändert an XML-XQL weitergeleitet.

Transformation nach SQL Wie weiter oben schon dargelegt, besteht eine gewisse syntaktische Ähnlichkeit in der Notationsweise von XQL und SQL. Abstrahiert man von der Semantik, ist eine XQL-Query fast schon eine SQL-Query. Die Bereiche `SELECT`, `FROM` und `WHERE` sind bereits deutlich zu erkennen. Der Pfad, welcher vor der aufgehenden eckigen Klammer steht entspricht der `SELECT`-Klausel. Innerhalb der eckigen Klammern befinden sich die Bedingungen. Das entspricht der `WHERE`-Klausel. Tabellen, die in SQL bei `FROM` angegeben werden, gibt es in diesem Sinne bei XQL/XML nicht. Hierbei handelt es sich um die XML-Files, die durchsucht werden sollen.

Mit den Regular Expressions kann ein String komfortabel nach bestimmten Zeichen oder Zeichenketten durchsucht werden. ASIX unterscheidet von vornherein zwischen Queries mit und ohne Bedingungen, denn Queries ohne Bedingungen müssen nicht transformiert werden. Da Bedingungen immer in eckigen Klammern

Ausgabepfad	Filterbedingungen	XML-File(s)
/element1/element2	[/element1/element2 = 500 and /element1/element3 < 2000]	xmlfile.xml

Abbildung 5.2: Teile einer XQL-Query

stehen, kann eine Query darauf getestet werden. Die Regular Expression `/\[/`[3] überprüft den Querystring auf eine linke eckige Klammer.

Der *split*-Operator kann Zeichenketten nach bestimmten Mustern zerteilen. In diesem Fall findet eine Teilung jeweils beim Auftreten eines Leerzeichens statt. Die einzelnen Teilzeichenketten werden in ein Feld geschrieben, wobei jede Teilkette ein Feldelement füllt.

Listing 5.4: Auszug aus xql2sql.pm

```
38 my (@words) = split ( /\s/, $xqlquery );
```

Anschließend wird der Ausgabepfad vom ersten Bedingungspfad getrennt. Dies geschieht wiederum mit der *split*-Anweisung, in der als Trennmuster eine aufgehende eckige Klammer angegeben ist.

Listing 5.5: Auszug aus xql2sql.pm

```
43 (@h1) = split ( /\[/, shift (@words) );
```

Die verschiedenen Teile einer Bedingung müssen an Sprite angepasst werden. So wird das optionale Quoting der logischen Operatoren `AND`, `OR` und `NOT` mit `$` entfernt. Die Vergleichsausdrücke werden durch ihre Synonyme in Zeichenform ersetzt (`ilt` durch <, `ile` durch <=, `igt` durch >, `ige` durch >= und `ie` durch =). Zur Verarbeitung in Sprite werden alle Slashes durch doppelte Unterstriche ersetzt, um sie von einfachen Unterstrichen zu unterscheiden. In XQL müssen Regular Expressions mit ' gequotet werden, in Sprite nicht. Deshalb wird dieses Quoting mit einer Substitutionsexpression entfernt.

Listing 5.6: Auszug aus xql2sql.pm

```
57 for ( my $i = 0 ; $i < scalar @words ; $i++ ) {
```

[3] Wegen der besonderen Bedeutung eckiger Klammern in Regular Expressions, muss sie in diesem Fall gequotet werden.

```
58      if ( $words[$i] =~ /(and|or|not)/i ) {
59          $words[$i] =~ s/\$//g;
60      }
61      $words[$i] =~ s/^ilt$/</g;
62      $words[$i] =~ s/^ile$/<=/g;
63      $words[$i] =~ s/^igt$/>/g;
64      $words[$i] =~ s/^ige$/>=/g;
65      $words[$i] =~ s/^ie$/=/g;
66
67      if ( ( $words[$i] =~ /^\// ) || ( $words[$i] =~ /^\(\// ) ) {
68          $words[$i] =~ s/^\///;
69          $words[$i] =~ s/^\(\//\(/;
70          $words[$i] =~ s/\//__/g;
71      }
72      if ( $words[$i] =~ /^'/␣)␣{
73          $words[$i] =~ s/'//g;
74      }
75
76  }
```

In einer FOR-Schleife werden zum Schluss alle Elemente zu einer gültigen SQL-Query zusammengefügt und an Sprite übergeben.

Listing 5.7: Auszug aus xql2sql.pm

```
81   my $sqlausgabe = "";
82  for (@words) {
83      $sqlausgabe .= $_ . "␣";
84  }
```

Queries ohne Bedingung Besteht eine Query nur aus dem Ausgabepfad, so wird dieser direkt zur Überprüfung auf einen vorhandenen Index benutzt. Danach werden die entsprechenden Komponenten aufgerufen.

Suche mit Sprite Die Verwendung von Komponenten spielt in der modernen Softwareentwicklung eine große Rolle. Anstatt einen Programmteil komplett neu zu entwickeln, ist es oft sehr hilfsreich, ein bestehendes Modul zu nutzten und dieses abzuändern oder, wie im Fall von Sprite, zu erweitern.

Bei Sprite wurde die Parameterübergabe optimiert, sowie die Ausgabe an ASIX angepaßt. Durch den dynamischen Aufbau der Indextabellen sind Spaltenanzahl und Breite ungewiß. Spezielle Routinen passen die Ausgabe an die

jeweilige Indextabelle an. Zum Beispiel sind so die Spaltenköpfe bei der Ausgabe immer genügend breit, so dass auch der breiteste Eintrag einer Spalte keine Verzerrungen in einer Zeile verursacht.

Die SQL-Klauseln `SELECT`, `FROM` und `WHERE` werden bei Parameterübergabe an das Modul übergeben. Mit den in der Bibliothek `Sprite.pm` vorhandenen Funktionen wird die SQL-Query verarbeitet. Anschließend erscheint das Ergebnis auf dem Bildschirm.

Suche mit XML-XQL Wenn noch kein Index zu einer Query existiert, wird die XQL-Query unverändert an XML-XQL weitergegeben. Jede Suche wird in einer Logdatei (Listing C.3 auf Seite 120) protokolliert, um später Daten für die Indexerstellung zu haben. Bei jeder Suche wird ein Eintrag an das Logfile angehängt. Dabei ist es unerheblich, ob schon ein Eintrag existiert oder nicht. Bei der Suche spielt lediglich die Performance eine große Rolle. Daher ist es günstiger, diese Datensätze, bestehend aus den *Pfaden der XQL-Query*, des Kalendertages des Jahres in Form einer Integerzahl[4] und der *Suchzeit* in Sekunden einfach an die Datei anzuhängen, als zu überprüfen, ob es schon ähnliche Einträge gibt, etc. In Abschnitt 5.3 wird erklärt wie die Daten der Logdatei zur Erstellung des Indexes verwendet werden. In dem Modul xml2txt.pm wird das Ergebnis der XQL-Suche für den Bildschirm aufbereitet und anschließend angezeigt.

5.3 Der Indexer

Die Erstellung der Indextabellen läuft unabhängig von den Suchanfragen. Sie kann entweder manuell oder über dnofity ausgelöst werden. Die Bedingungen zur Indexerstellung stehen in dem Konfigurationsfile und indirekt in dem Logfile. Nach der Auswertung beider Files ist klar, welche Indextabellen aufgebaut werden müssen. Das Modul kann über die Kommandozeile oder in dnofity gestartet werden. Der Aufruf sieht folgendermaßen aus:

```
$ asix.pl -i "XML-Pfad"
```

[4] entspricht dem laufenden Tag im Jahr ausgehend vom 1. Januar,
zum Beispiel 1. Februar = 32

Auf den Funktionsparameter folgt der Datenparameter "XML-Pfad". Dem Programm werden diese Parameter als Kommandozeilenparameter in dem Array `@ARGV` übergeben, wobei das erste Arrayelement auch der erste Parameter ist.

Auswertung des Konfigurationsfiles Die Auswertung des Konfigurationsfiles */etc/asix.conf* findet bei jedem Start von ASIX statt. Die Werte werden in globale Environmentvariable geschrieben, auf die der *Indexer* zugreifen kann.

Auswertung des Logfiles Bei jeder Suche mit XML-XQL innerhalb der Suchfunktion von ASIX wird, wie bereits in Kapitel 5.2 auf Seite 69 beschrieben, ein Eintrag der Logdatei */var/ASIX/logfile.lg* hinzugefügt. Eine Beispieldatei befindet sich in Anhang C.3 auf Seite 120. Da im Logfile jede ausgeführte XQL-Query gespeichert ist, können die gleichen Pfade auch mehrfach vorkommen. Zur Auswertung der Indexerstellungskriterien müssen aus dem Logfile für alle gleichen Pfadtupel, deren Anzahl und mittlere Suchzeiten bestimmt werden. Der im Logfile protokollierte Kalendertag des Jahres kann für erweiterte Funktionen genutzt werden (zum Beispiel, um ältere Indexfiles zu löschen). Durch das Format des Logfiles können die einzelnen Bestandteile mit der *split*-Funktion getrennt werden, wobei die Variablen `$oneline` eine Zeile des Logfiles, `$p` ein Pfadtupel und `$t` die Suchzeit enthalten.

```
55 my ( $p, $d, $t ) = split ( /\t/, $oneline );
```

Die einzelnen, von einander verschiedenen Pfadtupel werden in dem dreidimensionalen Array *path_num_time* gespeichert. Es hat die Form:

```
[Pfadtupel][Anzahl des Auftretens im Logfile][Suchzeiten]
```

Die Pfadtupel und Suchzeiten sind jeweils durch Leerzeichen voneinander getrennt. Ist ein Pfadtupel noch nicht im Array *path_num_time* enthalten, wird es als neues Element zusammen mit der Suchzeit hinzugefügt. Die Anzahl des Auftretens wird auf `1` gesetzt. Ist ein Pfadtupel bereits im Array *path_num_time* enthalten, wird die Anzahl des Auftretens um eins erhöht und die Suchzeit durch ein Leerzeichen an das dritte Element angehängt.

```
62 for ( my $k = 0 ; $k <= $number ; $k++ ) {
63     if ( $p eq $path_num_time[$k][0] ) {
64         $match = "YES";
65         $path_num_time[$k][1]++;
66         $path_num_time[$k][2] .= "␣" . $t;
67     }
68 }
69 if ( $match ne "YES" ) {
70      $path_num_time[$number][0] = $p;
71      $path_num_time[$number][1] = 1;
72      $path_num_time[$number][2] = $t;
73      $number++;
74 }
```

Mit der Mittelwertfunktion wird der Mittelwert der Einzelsuchzeiten berechnet und im dritten Element gespeichert. Anzahl und Mittelwert werden nun mit den Parametern aus der Konfigurationsdatei verglichen. Wird einer dieser Parameter überschritten, wird ein Index erstellt. Das kann zum einen passieren, wenn die mittlere Suchzeit einer Query den `MIN_SEARCHTIME`-Wert überschreitet oder die minimale Anzahl von Queryaufrufen, die in der Variable `MIN_QUERYCALLS` gespeichert ist, überschritten wurde.

Das vollständige Script von indexer.pm ist in Anhang C.4 auf Seite 121 nachzulesen.

Indextest Wie bei den XQL-Anfragen muss auch hier geprüft werden, ob eventuell schon Indexfiles existieren. Mit Hilfe der Funktion *build_index_file_name* werden aus den Pfaden die Dateinamen der Indextabellen gebildet. Anschließend wird getestet, ob bereits Indexfiles mit diesen Namen existieren. Wenn ja, braucht für diese Datei kein neuer Index erstellt werden.

Aufbau eines Indexfiles Ein Indexfile besteht immer aus dem Tabellenkopf und den Daten. Die erste Spalte ist für den Dateinamen der Werte aus den restlichen Spalten reserviert. Jeder Pfad hat eine eigene Spalte (Listing 5.8 auf der folgenden Seite. Somit beträgt die Anzahl der Spalten einer Indexdatei immer:

Formel 5.3.1

Anzahl der Spalten = Anzahl der verschiedenen Pfade einer XQL-Query + 1

Da die Pfade auch aus den Bedingungen extrahiert wurden, fließen diese auch nicht direkt in den Index mit ein. Es wird lediglich auf die Existenz der XML-Tags der Pfade geprüft, nicht aber auf deren Inhalt. Deswegen werden auch in einem Index alle möglichen Werte eines Pfadtupels gespeichert.

$$(Pfad_1, \ldots, Pfad_n)$$

$$n = \text{Alle möglichen Pfade innerhalb eines XML-Files}$$

Die Anzahl der Werte beträgt daher:

Formel 5.3.2

$$Anzahl\ der\ Zeilen \leq$$

$$Inhalt(Pfad_1(XML - File_1)), \ldots, Inhalt(Pfad_n(XML - File_1))$$

$$\ldots$$

$$Inhalt(Pfad_1(XML - File_m)), \ldots, Inhalt(Pfad_n(XML - File_m))$$

$$n, m \in N$$

Das heißt, ein Index enthält so viele Einträge, wie es XML-Files mit gültigen Pfaden gibt.

Listing 5.8: group__assistant__name__lastname.group__assistant__private__age.idx

```
1 Filename group__assistant__name__lastname group__assistant__private__age
2 __home__hotzky__asix__files__test1.xml Smith      26
3 __home__hotzky__asix__files__test2.xml Hackleton  23
4 __home__hotzky__asix__files__test3.xml Regnault   18
```

Beispiel 5.3.1 *In Listing 5.8 wird angenommen, dass die Datensammlung drei XML-Files enthält. Der* Indexer *hat festgestellt, dass für die Pfade*

`/group/assistant/name/lastname`

und

`/group/assistant/private/age`

ein Index erzeugt werden muss. Dazu wird die XML-Datenbank nach diesen Pfaden durchsucht und alle Ergebnisse werden in das File eingetragen.

Erzeugung eines Indexfiles Stellt der Indexer fest, dass ein Index erzeugt werden muss, werden genau soviele XQL-Anfragen erzeugt, wie es Pfade gibt, wobei jeder Pfad genau einmal der Ausgabepfad ist, während die anderen in der Filterbedingung mit `AND`-Verknüpfungen auf Existenz überprüft werden. Da bei der Ausgabe die Reihenfolge der Ergebnisse immer gleich bleibt, kann jedes Ergebnis im Hauptspeicher gehalten werden. Anschließend werden alle Ergebnisse zu einer neuen Tabelle zusammengefügt und als neue Indexdatei geschrieben. Beispiel 5.3.2 zeigt den Aufbau der Queries zur Indexerzeugung. Existieren in einem Pfadtupel zwei identische Pfade, so werden diese bei der Querybildung als einer angesehen. Dieser Algorithmus wird für alle zu erstellenden Indextabellen angewendet.

Beispiel 5.3.2 *XQL-Queries bei der Indexerstellung*

1. Anfrage: $pfad_1[pfad_2\ \$AND\$\ \ldots\ \$AND\$\ pfad_n]$

2. Anfrage: $pfad_2[pfad_3\ \$AND\$\ \ldots\ \$AND\$\ pfad_n\ \$AND\$\ pfad_1]$

$\vdots$

n. Anfrage: $pfad_n[pfad_1\ \$AND\$\ \ldots\ \$AND\$\ pfad_{n-1}]$

5.4 Das Update

Der Index einer Datenbank steht immer in direktem Zusammenhang mit den Originaldaten. Sobald diese sich ändern, ist der Index nicht mehr aktuell und damit unvollständig. Ein gutes Datenbanksystem, das die Vorteile eines Indexes nutzt, muss deswegen auch immer Funktionen bereitstellen, die den Index möglichst in Echtzeit aktualisieren. Mit *dnotify* kann die Aktualisierung im Millisekundenbereich ausgelöst werden. Danach ist die Zeit der Aktualisierung nur noch von der Anzahl der betroffenen Indexfiles und der damit verbundenen Anzahl an XQL-Queries abhängig. Da die XQL-Suche verhältnismäßig lange dauert, wurde ein Algorithmus entwickelt, der in der zu aktualisierenden Indexdatei den Eintrag, der ersetzt werden muss, heraussucht und ihn aktualisiert. Somit kann die Suchzeit mit XQL minimiert werden, und der Index ist in kürzester Zeit wieder aktuell.

Updatealgorithmus:

1. Das betroffene XML-File muss bestimmt werden.
2. Alle Indexfiles müssen auf das Enthaltensein des gefundenen XML-Files hin überprüft werden.
3. Wurde in einem Indexfile ein Eintrag der XML-Datei gefunden wird dieser gelöscht.
4. Für das Hinzufügen, Ändern und Löschen existieren verschiedene Teilschritte
   - Hinzufügen/Ändern:
     - Aus dem Namen des Indexfiles werden nach dem Muster zur Indexerstellung XQL-Queries gebildet.
     - Die XQL-Suche wird nur auf das eine Indexfile angewendet.
     - Das Ergebnis wird an das vorhandene Indexfile angehängt.
   - Löschen: Das XML-File wird vom Datenträger gelöscht.

5.4.1 Hinzufügen und Ändern

dnotify Das im Hintergrund laufende *dnotify* führt einen Befehl (`dnotify -e`) aus, sobald in dem zu prüfenden Verzeichnis `$xmlpath` eine Datei geändert wurde oder eine neue Datei abgespeichert wurde (`dnotify -M`). Damit kann jede Änderung in den Original-XML-Dokumenten registriert werden. Der Befehl, der ausgeführt wird, wenn eine Änderung erkannt wurde, ist der Aufruf von ASIX mit dem Parameter `asix.pl -u`.

```
dnotify -M $xmlpath -e perl asix.pl -u $xmlpath
```

dnotify wird in einem ShellScript gestartet. Da es nur meldet, dass sich etwas geändert hat, aber nicht welche Datei, muss der Zustand des Verzeichnisses vor der Änderung gespeichert werden. Dazu wird ein `ls` aller XML-Files des Verzeichnisses in eine temporäre Datei umgeleitet.

```
ls -l $1*.xml > /var/asix/ls.dat
```

Filebestimmung Nachdem aus *dnotify* heraus ASIX gestartet wurde, muss bestimmt werden, welche Datei den Aufruf ausgelöst hat. Dazu werden die Daten des alten `ls` mit denen eines neuen `ls` verglichen. Mit dem Befehl `find -newer` läßt sich erkennen, welche Dateien im neuen `ls` neuer sind als die im alten `ls`. Wenn die Datei gefunden wurde, wird `ls.dat` mit den Daten des neuen `ls` überschrieben und beim nächsten Auslösen von *dnotify* verwendet.

```
62 my $found_xml_file = `find $ENV{XML_PATH} -newer $ENV{VAR_PATH}ls.dat`;

64 my $x       = `ls -l $ENV{XML_PATH} > $ENV{VAR_PATH}ls.dat`;
```

Aktualisierung einer Indexdatei Um herauszufinden, welche Indexfiles zu aktualisieren sind, müssen alle Indexfiles der Reihe nach geöffnet werden. Befindet sich im einem File ein Eintrag mit dem gefundenen geänderten File, wird dieser Eintrag aus der Indextabelle gelöscht. Aus dem Dateinamen des betroffenen Indexfiles werden nun die entsprechenden XQL-Anfragen erzeugt. Bei der Suche mit XML-XQL werden nun nicht alle Indexfiles angegeben, sondern nur das eine, das sich geändert hat. Das Ergebnis aller Anfragen wird anschließend an die Indexdatei angehängt. Somit ist die Suchzeit minimal, da nicht die ganze Indexdatei neu erzeugt werden muss.

5.4.2 Löschen

Der Index muss auch erneuert werden, wenn ein XML-File gelöscht wurde. Dies kann entweder über ein zweites, im Hintergrund laufendes, *dnotify* geschehen oder manuell. Bei der automatisierten Variante, wird *dnotify* mit der Option `-D` gestartet, das ASIX mit dem Parameter `-d` aufruft, sobald eine Datei aus dem Verzeichnis, das geprüft wird, gelöscht wurde:

```
dnotify -D $xmlpath -e perl asix.pl -d $xmlpath
```

Mit einem Vergleich zweier `ls` vor und nach dem Löschen, wird die gelöschte Datei bestimmt. Wie bei der Aktualisierung der Indexdateien, muss jede Indexdatei geöffnet werden. Wird ein Eintrag mit der gelöschten XML-Datei gefunden, wird dieser aus der Indextabelle entfernt.

Läuft kein zweites *dnotify* auf dem Rechner, kann eine Datei auch manuell gelöscht werden. Dazu wird ASIX mit `asix -d` aufgerufen und dem Namen der zu löschenden Datei.

```
perl asix.pl -d delete.xml
```

Kapitel 6

Evaluierung der Implementierung

In der Softwareentwicklung müssen möglichst performante und zuverlässige Softwaresysteme erstellt werden. Eine der wichtigsten Methoden zur Qualitätssicherung sind umfangreiche Tests und Benchmarks. Durch verschiedene Testreihen kann so das Softwareprodukt nach Schwachstellen untersucht werden, die anschließend durch das wiederholte Durchlaufen der Phasen *Design* und *Implementierung* (siehe Abb. 4.1 auf Seite 51) verbessert werden können[1]. Außerdem kann mit einer Evaluierung überprüft werden, ob das entwickelte System den Anforderungen, die an die Entwicklung gestellt wurden, gerecht wird.

6.1 Die Testumgebung

Das Softwaresystem ASIX kann auf allen Umgebungen eingesetzt werden, auf denen Perl eingesetzt werden kann. Für die Testreihen wurde ein SONY VAIO Notebook (Intel Pentium III 840 MHz Prozessor, 256 MB Hauptspeicher, Red Hat Linux 8.0 3.2-7 Betriebssystem) verwendet. Mit einem Perlscript wurden verschiedene XML-Testdatenbanken generiert. Die folgenden Listings zeigen Ausschnitte aus den generierten XML-Files[2]:

[1] Bei groben Fehlern im System kann es auch nötig sein die Phasen *Anforderungen* und *Analyse* nochmals zu durchlaufen

[2] Jeweils ein vollständiges XML-File befindet sich im Anhang C.5 auf Seite 160

Listing 6.1: Auszug aus Listing C.13 auf Seite 161

```
<?xml version='1.0'?>
<root>
     <element1>1000</element1>
     <element2>2000</element2>
     <element3>3000</element3>
:
</root>
```

Listing 6.2: Auszug aus Listing C.14 auf Seite 161

```
<?xml version='1.0'?>
<root>
     <element1>
          <element11>1000</element11>
          <element12>2000</element12>
     </element1>
:
</root>
```

Listing 6.3: Auszug aus Listing C.15 auf Seite 162

```
<?xml version='1.0'?>
<root>
     <element1>
          <element11>
               <element111>4000</element111>
               <element111>5000</element111>
          </element11>
          <element12>2000</element12>
     </element1>
:
</root>
```

Listing 6.4: Auszug aus Listing C.16 auf Seite 163

```
<?xml version='1.0'?>
<root>
     <element1>
          <element11>
               <element111>
                    <element1114>3000</element1114>
                    <element1115>6000</element1115>
               </element111>
               <element111>5000</element111>
          </element11>
          <element12>2000</element12>
     </element1>
```

```
13 :
14 </root>
```

Jeweils Anfragen wie in den Beispielen 6.1.1 bis 6.1.4 mit ein, zwei, drei und vier Bedingungen wurden gewählt, um die Leistung und Performance von ASIX zu testen.

Beispiel 6.1.1 `/root/element1[/root/element1 < 2000]`

Beispiel 6.1.2
```
/root/element1[/root/element1 < 2000
              $and$ /root/element3 > 2000]
```

Beispiel 6.1.3
```
/root/element1[/root/element1 < 2000
              $and$ /root/element3 > 2000
              $and$ /root/element9 > 8000]
```

Beispiel 6.1.4
```
/root/element1[/root/element1 < 2000
              $and$ /root/element3 > 2000
              $and$ /root/element9 > 8000
              $and$ /root/elementC =~ '/CCCC/']
```

6.2 Testreihe 1 - Suchaufwand ohne Index

In der ersten Testreihe wurde der Zeitaufwand von ASIX bei der Suche in XML-Files ohne die Verwendung von Indextabellen getestet. Untersucht wurde die Abhängigkeit der Suchzeiten von der Verschachtelungstiefe der XML-Files, deren Anzahl und der Anzahl der Bedingungen in einer XQL-Query. Alle Messwerte können in Anhang A.2.1 auf Seite 105 eingesehen werden.

6.2.1 Abhängigkeit der Suchzeiten von Querybedingungen

Die Suchzeiten bei unterschiedlicher Anzahl von Bedingungen in einer XQL-Query weichen nur geringfügig voneinander ab. Bis 200 XML-Files befinden sich diese Zeitunterschiede im Millisekundenbereich. Erst bei 2000 XML-Files sind Unterschiede im Sekundenbereich zu erkennen. Somit ist das Antwortzeitverhalten nicht abhängig von der Anzahl der Querybedingungen. Abbildung 6.1 auf der folgenden Seite verdeutlicht graphisch diese Erkenntnis.

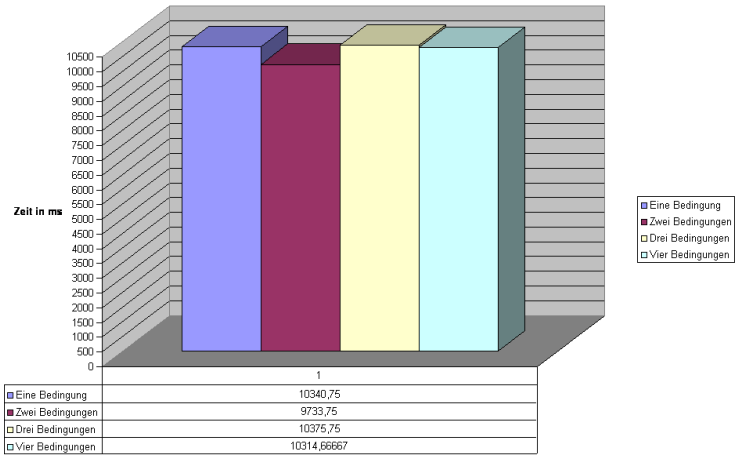


Abbildung 6.1: ASIX-Antwortzeiten ohne Index (nach Bedingungen)

6.2.2 Abhängigkeit der Suchzeiten von der Verschachtelungstiefe

Die Suchzeiten bei unterschiedlicher Verschachtelung der XML-Files verhalten sich ähnlich wie bei unterschiedlicher Anzahl von Bedingungen. Auch hier liegen die Werte im Millisekundenbereich. Alle Werte in Abbildung 6.2 stammen aus Anfragen auf 200 XML-Files. Wie auch in Tabelle A.2.1 auf Seite 105 zu erkennen ist, unterscheiden sich die Werte bei der Suche mit 2000 Files auch nur geringfügig. Die Antwortzeiten von ASIX sind daher nicht von der Verschachtelungstiefe der XML-Files abhängig.

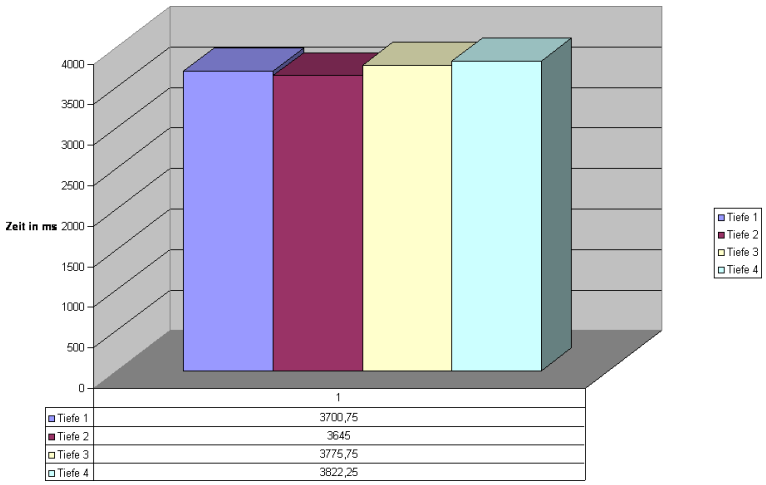


Abbildung 6.2: ASIX-Antwortzeiten ohne Index (nach Verschachtelungstiefe)

6.2.3 Abhängigkeit der Suchzeiten von der Anzahl der XML-Files

Abschließend wurden die Suchzeiten in Abhängigkeit der Anzahl zu durchsuchender XML-Files gemessen. Hier sind die Unterschiede deutlich größer als in den vorhergehenden Vergleichen. So steigen die Antwortzeiten bei 200 auf das ca. Dreifache gegenüber den bei 20 XML-Files. Bei 2000 XML-Files ist die XQL-Suche ca. 7-mal langsamer als bei 20 Files. Abbildung 6.3 verdeutlicht dies graphisch.

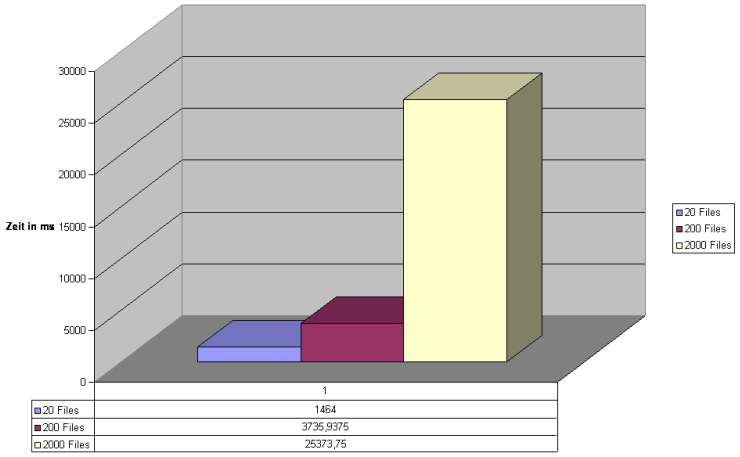


Abbildung 6.3: ASIX-Antwortzeiten ohne Index (nach XML-Files)

6.3 Testreihe 2 - Suchaufwand mit Index

In der zweiten Testreihe wurde der Zeitaufwand von ASIX bei der Suche in XML-Files mit der Verwendung von Indextabellen getestet. Untersucht wurde die Abhängigkeit der Suchzeiten von der Verschachtelungstiefe der XML-Files, deren Anzahl und der Anzahl der Bedingungen in einer XQL-Query. Alle Messwerte können in Anhang A.2.2 auf Seite 106 eingesehen werden.

6.3.1 Abhängigkeit der Suchzeiten von Querybedingungen

Im Verhältnis zur Gesamtsuchzeit sind hier größere Unterschiede festzustellen. So vergrößert sich die Antwortzeit je Bedingung um ca. 12%. Jedoch selbst bei 2000 XML-Files liegen die gemessenen Werte unter einer Sekunde (siehe Abbildung 6.4).

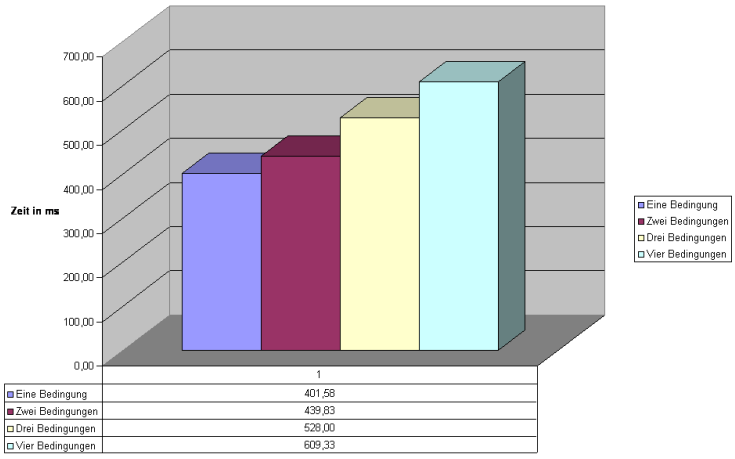


Abbildung 6.4: ASIX-Antwortzeiten mit Index (nach Bedingungen)

6.3.2 Abhängigkeit der Suchzeiten von der Verschachtelungstiefe

Die Suchzeiten bei unterschiedlicher Verschachtelung der XML-Files weichen kaum voneinander ab. Sie bewegen sich um den Bereich von 300 Millisekunden. Alle Werte in Abbildung 6.5 auf der nächsten Seite stammen aus Anfragen auf 200 XML-Files. Die Antwortzeiten von ASIX sind daher auch mit Index nicht von der Verschachtelungstiefe der XML-Files abhängig.

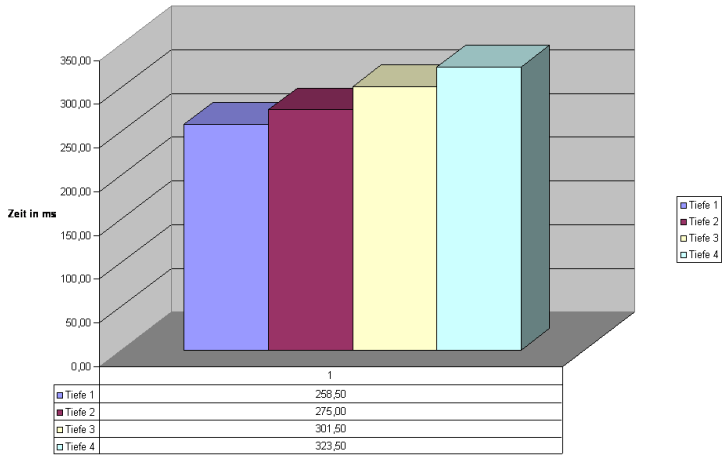


Abbildung 6.5: ASIX-Antwortzeiten mit Index (nach Verschachtelungstiefe)

6.3.3 Abhängigkeit der Suchzeiten von der Anzahl der XML-Files

Wie zu erwarten war, sind die Zeiten auch bei der Suche mit Index größer, wenn die Zahl der zu durchsuchenden XML-Files höher wird. Dies ist darauf zurückzuführen, dass auch die Indexfiles länger werden. Der Mittelwert bei 2000 XML-Files liegt hier bei ca. 970 ms (siehe auch Abbildung 6.6).

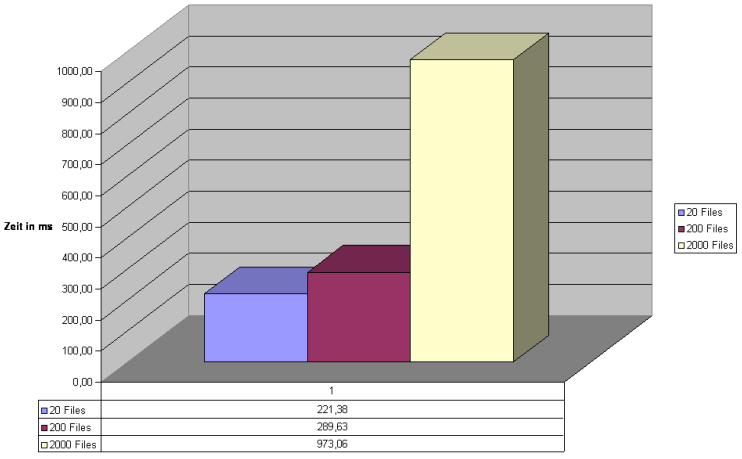


Abbildung 6.6: ASIX-Antwortzeiten mit Index (nach XML-Files)

6.4 Performancegewinn durch Indexe

In den Charts 6.7 bis 6.9 ab Seite 90 ist deutlich der Performancegewinn durch die Verwendung einer Indexdatenbank zu erkennen. Auch hier sind wieder die Zeiten für die drei Fälle, XML-Files, Verschachtelungstiefe und Anzahl der Bedingungen, angegeben. Die geringeren Zeiten sind auf den schnellen Suchalgorithmus von Sprite und die Verarbeitung der Anfragen in ASIX zurückzuführen. Selbst bei einer kleinen Anzahl von XML-Files lohnt sich bereits die Erstellung von Indextabellen. Da die Größe des Indexes nur einen Bruchteil der Originaldaten beträgt, stellt dies auch keinen schwerwiegenden Verlust an Plattenspeicherkapazität dar. Die erstmalige Erstellung einer Indexdatenbank erfordert aufgrund der vielen XQL-Queries einen relativ hohen Zeitaufwand gegenüber den Suchzeiten. Da dies jedoch auch in Form einer Task zum Beispiel nachts durchgeführt werden kann, stellt es kein Problem dar. Die Zeiten hierfür lagen zwischen 15 Sekunden und 5 Minuten.

Der Performancegewinn steigt mit der Größe der XML-Datenbank. Mit der Testdatenbank konnte ein maximaler Performancegewinn von 44,5 erzielt werden. Der mittlere Verbesserungsfaktor liegt bei 16. Abbildung 6.10 auf Seite 92 zeigt wie die Suchzeiten mit ASIX im Durchschnitt verbessert werden konnten.

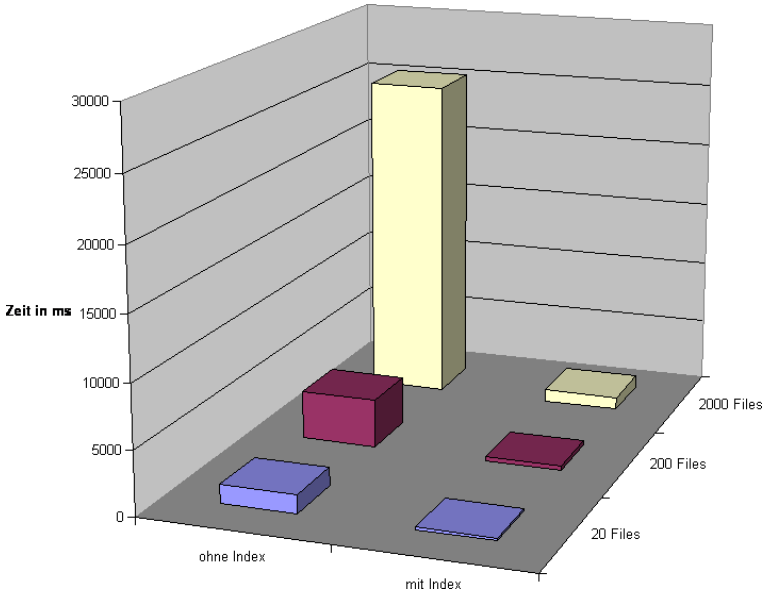


Abbildung 6.7: Zeitunterschiede mit und ohne Index (nach XML-Files)

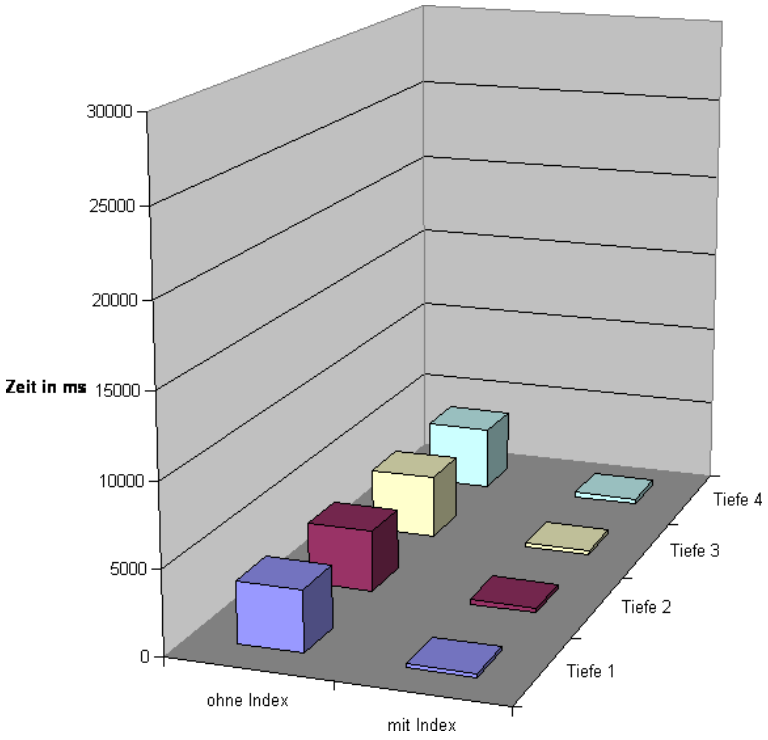


Abbildung 6.8: Zeitunterschiede mit und ohne Index (nach Verschachtelungstiefe)

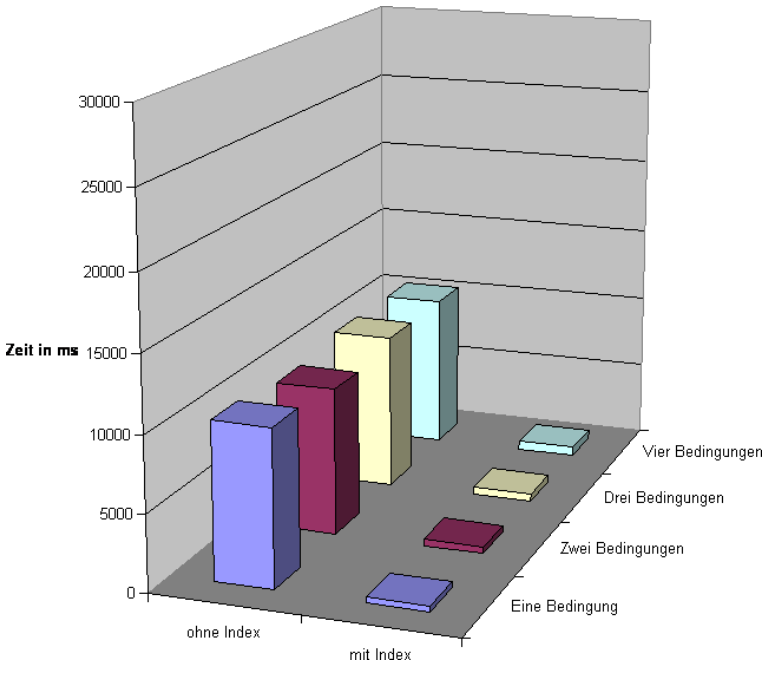
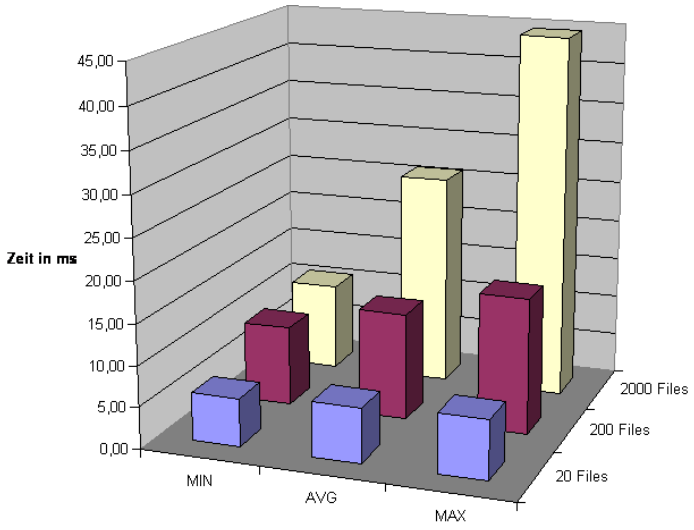


Abbildung 6.9: Zeitunterschiede mit und ohne Index (nach Bedingungen)



	MIN	AVG	MAX
20 Files	5,81	6,62	7,25
200 Files	9,75	13,04	16,53
2000 Files	10,66	26,00	44,50

Abbildung 6.10: Minimaler, mittlerer und maximaler Verbesserungsfaktor

Kapitel 7

Zusammenfassung

7.1 Entwicklungsstand

Das in dieser Arbeit entwicklelte selbstadaptierende Suchverfahren in leichtgewichtigen XML-Datenbanken semistrukturierter Daten berücksichtigt gegenüber anderen Konzepten in starken Maße die Historie bereits gestellter Anfragen an XML-Datenbanken. Die Überlegungen, ein selbstadaptierendes Suchverfahren zu entwickeln, haben dazu geführt, bereits gestellte Anfragen zu protokollieren. Durch die Protokollierung aller Suchanfragen und die daraus erzeugten Indexfiles lassen sich zukünftige Anfragen beschleunigen. Eine weitere Idee war, die Ergebnisse bereits getätigter Anfragen so zu speichern und aufzubereiten, dass bei späteren Anfragen darauf zugegriffen werden kann und diese ohne den aufwendigen XQL-Suchalgorithmus bearbeitet werden können.

Das in Umsetzung dieses Verfahrens entwickelte Softwaresystem ASIX entstand in einem iterativen Prozess.

Für ASIX wurde ein Indexfile-Konzept entwickelt, das aufgrund des Protokolls bereits gestellter Anfragen eine Indexdatenbank erzeugt, die aus einer Ansammlung von ASCII-Tabellen besteht. Die Protokolldatei enthält die Pfadtupel und Suchzeiten vorangegangener Anfragen. Eine Indextabelle wird dann erzeugt, wenn der Mittelwert der Suchzeiten oder die Häufigkeit jedes gleichen Pfadtupels einen vorab angegebenen Wert übersteigt. So enthält jede Datei die Werte aller XML-Files eines Pfadtupels, wobei jede Spalte einen Pfad repräsentiert.

ASIX wurde protypisch für die Suchanfrage in XQL realisiert. Anfragen an die

XML-Datenbank werden immer in XQL formuliert, auch dann, wenn bereits ein Indexfile, in dem in SQL gesucht werden muss, existiert. Der Performancegewinn konnte vor allem dadurch erreicht werden, dass die Suchzeiten in den Indextabellen, trotz der notwendigen Konvertierung von XQL nach SQL, deutlich geringer sind als mit dem XQL-Algorithmus.

Mit diesem System wird die Lücke zwischen SQL-Datenbanken und Volltextsuche geschlossen.

7.2 Ausblick

ASIX läßt sich aufgrund seiner einfachen Nutzerschnittstelle sehr leicht in bereits vorhandene Systeme integrieren. Denkbar ist auch die Erstellung einer graphischen Weboberfläche, die es einem Nutzer erlaubt, weltweit auf den Datenbestand zuzugreifen. Mit einer dynamischen Oberfläche wäre es möglich, eine Webseite zu erstellen, die sich der jeweiligen XML-Datenbank anpaßt. Somit wäre ein Anwender in der Lage ASIX zu benutzen ohne dabei XQL zu kennen, indem er die Pfade und die logischen Operatoren in Dropdown-Menüs auswählen kann.

Nach einer längeren Testphase wäre es denkbar, ASIX in das CPAN Repository einzustellen und damit einem breiten Nutzerkreis zugänglich zu machen.

Literaturverzeichnis

[ABC+98] V. Apparao, S. Byrne, M. Champion, S. Isaacs, I. Jacobs, A. L. Hors, G. Nicol, J. Robie, R. Sutor, C. Wilson, L. Wood. *Document Object Model (DOM) Level 1 Specification* [1998].
URL `http://www.w3.org/TR/REC-DOM-Level-1/`

[ABK00] R. Anderson, M. Birbeck, M. Kay. *XML Professionell* (MITP-Verlag, 2000). ISBN 3-8266-0633-7.
URL `http://www.wrox.com`

[AQM+97] S. Abiteboul, D. Quass, J. McHugh, J. Widom, J. Wiener. *The lorel query language for semistructured date. International Journal on Digital Libraries*, Band 1:Seite 68ff [1997].

[BM00] H. Behme, S. Mintert. *XML in der Praxis* (Addison-Wesley, 2000). ISBN 3-8273-1636-7.

[CFMR01] D. Chamberlin, P. Fankhauser, M. Marchiori, J. Robie. *XML Query Requirements* [2001].
URL `http://www.w3.org/TR/xmlquery-req`

[CGH+94] S. Chawathe, H. Garcia-Molina, J. Hammer, K. Ireland, Y. Papakonstantinou, J. Ullman, J. Widom. *The TSIMMIS Project: Integration of Heterogeneous Information Sources. Proceedings of IPSJ Conference*, (Seite 7 bis 18) [1994].

[Cod03] E. Codd. *The Relational Model for Database Management: Version 2* (Addison-Wesley, 2003).

[com01] *Computer Lexikon Fachwörterbuch* (Microsoft Press, 2001). ISBN 3-86063-824-6.

[cpa03] *CPAN* [2003].
URL `http://www.cpan.org`

[CRF00] D. Chamberlin, J. Robie, D. Florescu. *Quilt: An xml query language for hetergeneous data sources. International Workshop on the Web and Databases (WebDB), Dallas* [2000].

[Exc03] *Extensible Information Server* [2003].
URL `http://www.exeloncorp.com`

[exe03] *Extensible Information Server* [2003].
URL `http://www.exeloncorp.com`

[FFLS00] M. Fernandez, D. Florescu, A. Levy, D. Suciu. *Declarative Specification of Web Sites with Strudel. VLDB Journal* [2000].

[Fri98] J. Friedl. *Reguläre Ausdrücke* (O'Reilly, 1998). ISBN 3-930673-62-2.

[FSW99] M. Fernandez, J. Siméon, P. Wadler. *XML Query Languages: Experiences and Exemplars* [1999].
URL `http://www.w3.org/1999/09/ql/docs/xquery.html`

[Gle03] A. Glew. *Perl-SQL* [2003].
URL `http://www.cs.wisc.edu/ glew/perl-sql-brief.html`

[GR95] R. Gabriel, H. Röhrs. *Datenbanksysteme* (Springer, 1995). ISBN 3-540-60079-5.

[Haj98] F. Hajji. *Perl, Einführung, Anwendung, Referenz* (Addison-Wesley, 1998). ISBN 3-8372-1335-x.

[Her96] H. Herold. *UNIX-Shells* (Addison-Wesley, 1996). ISBN 3-8273-1035-0.

[HR01] T. Härder, E. Rahm. *Datenbanksysteme* (Springer, 2001). ISBN 3-540-42133-5.

[HS03] D. Hollander, C. Sperberg-McQueen. *Happy Birthday, XML!* [2003]. URL `http://www.w3.org/2003/02/xml-at-5.html`

[Hug03] *Hughes Technologies* [2003]. URL `http://www.hughes.com.au`

[HV02] C. Hille, D. Völzke. *Zwei neue Entwicklungen: XML-Schema und XQuery* [2002].

[ISO86] *ISO8879 – Information processing – Text and Office Systems – Standard Generalized Markup Language (SGML)* [1986]. URL `http://www.iso.ch/cate/d16387.html`

[Jun01] P. Jung. *KDE 2.2 Detailverliebt. Linux-Magazin*, Band 10 [2001]. URL `http://www.linux-magazin.de/Artikel/ausgabe/2001/10/kde22/kde.html`

[Klo02] S. Klopp. *Automatische Generierung von virtuellen XML-Sichten aus relationalen Datenbankschemata und Übersetzung von XQuery-Anfragen nach SQL. Tagungsband zum 14. GI-Workshop Grundlagen von Datenbanken, Strandhotel Fischland, Halbinsel Fischland-Darß-Zingst, Mecklenburg-Vorpommern, 21. bis 24. Mai 2002. (CS-01-02) Fachbereich Informatik, Universität Rostock 2002* [2002].

[KM03] M. Klettke, H. Meyer. *XML & Datenbanken* (dpunkt.verlag, 2003). ISBN 3-89864-148-1.

[Kö00] D. Köhler. *XML - Query Sprachen, Studentische Ausarbeitung zum Seminar 'Datenintegration'* [2000].

[Kre01] K. Krebs. *XML und Datenbanken - Abfragesprachen* [2001].

[LH02] T. Leich, H. Höpfner. *Konzeption eines Anfragesystems für leichtgewichtige, erweiterbare DBMS. Tagungsband zum Workshop "Mobile Datenbanken und Informationssysteme - Datenbanktechnologie überall und jederzeit"*, (Seite 33 bis 37) [2002].

[LHW+00] A. Le Hors, P. L. Hégaret, L. Wood, G. Nicol, J. Robie, M. Champion, S. Byrne. *Document Object Model (DOM) Level 2 Core Specification* [2000].

[MAG+97] J. McHugh, S. Abiteboul, R. Goldman, D. Quass, J. Widom. *Lore: A Databank Management System for semistructured Data. SIGMOD Record*, Band 26(3):Seite 54 bis 66 [1997].

[MAG+00] J. McHugh, S. Abiteboul, R. Goldman, D. Quass, J. Widom. *Lore: A Databank Management System for semistructured Data* [2000].

[MAM+98] G. Mecca, P. Atzeni, A. Masci, P. Merialdo, G. Sindoni. *The Araneus Web-Based Management System. Exhibits Program of ACM SIGMOD* [1998].

[Map03] *Maple Design* [2003].
URL `http://www.mapledesign.co.uk/design.php`

[Mel98] S. Melnik. *Architektur und Realisierung eines verteilten XML-basierten Informationssystems* [1998].

[Mod01] Modulepool. *Ein einfacher Einstieg in die Syntax von XML* [2001].
URL `http://www.modulepool.com/web-pool/XML/syntax.php4#nummer6`

[Mor02] M. Morrison. *Sams Teach Yourself XML in 24 Hours* (Sams Publishing, 2002). ISBN 0-672-32213-7.

[MW93] U. Manber, S. Wu. *Glimpse: A Tool to Search through entire File Systems* [1993].

[mys03] *MySQL* [2003].
URL `http://www.mysql.com`

[Orw02] J. Orwant, Herausgeber. *Computer Science & Perl Programming* (O'Reilly, 2002). ISBN 0-596-00310-2.

[Pei03] R. Peine. *Objektorientierung und große XML-Dateien* [2003].
URL `http://www.perl-workshop.de/2003/contriblist.epl#132`

[PGMW95] Y. Papakonstantinou, H. Gracia-Molina, J. Widom. *Object Exchange Across Heterogeneous Information Sources. Proceddings of the IEEE International Conference on Data Engineering* [1995].

[RM02] E. Ray, J. McIntosh. *Perl & XML* (O'Reilly, 2002). ISBN 3-89721-148-3.

[Roy70] W. Royce. *Managing the Delevopment of large Software Systems, Concepts and Techniques* (IEEE Wescon, 1970).

[RV03] E. Rahm, G. Vossen. *Web und Datenbanken* (dpunkt.verlag, 2003). ISBN 3-89864-189-9.
URL `http://www.dpunkt.de`

[Sed92] R. Sedgewick. *Algorithmen* (Addison-Wesley, 1992). ISBN 3-89319-402-9.

[sgm03] *Getting started with SGML* [2003].
URL `http://www.arbortext.com/data/getting_started_with_SGML/getting_started_with_sgml.html`

[SH99] G. Saake, A. Heuer. *Datenbanken: Implementierungstechniken* (MITP, 1999). ISBN 3-8266-0513-6.

[sha03] *Shakespeare in XML* [2003].
URL `http://www.ibiblio.org/xml/examples/shakespeare/`

[Sof03] SoftwareAG. *Software AG: Informationen zu XML* [2003].
URL `http://www.softwareag.com/xml/dt/einstieg.htm`

[SP02] R. Schwartz, T. Phoenix. *Einführung in Perl* (O'Reilly, 2002). ISBN 3-89721-147-5.

[Spi88] T. Spitta. *Softwareengineering und Prototyping* (Springer, 1988). ISBN 3-540-17542-3.

[spr03a] *Cpan: Sprite* [2003].
URL `http://search.cpan.org/search?dist=Sprite`

[spr03b] *Sprite.pm: Extra Information* [2003].
URL `http://www.insite.com.br/rodrigo/work/sprite.html`

[Sta02] B. Stadler. *Anfragesprache für ein sematisches Netz* [2002].

[Ste98] R. Stephens. *SQL in 21 Tagen* (Markt & Technik, 1998). ISBN 3-8272-2020-3.

[SW02] D. Sparling, F. Wiles. *Perlmodule* (Addison-Wesley, 2002). ISBN 3-8273-1908-0.

[SW03] J. Siméon, P. Wadler. *The Essence of XML. The 30th Annual ACM SIGPLAN - SIGACT Symposium on Principles of Programming Languages New Orleans, LA (POPL'2003)* [2003].
URL `http://www-db.research.bell-labs.com/user/simeon/xml-essence.pdf`

[tam03] *Tamino* [2003].
URL `http://www.tamino.com`

[Vos94] G. Vossen. *Datenmodelle, Datenbanksprachen und Datenbank-Management-Systeme* (Addison-Wesley, 1994). ISBN 3-89319-566-1.

[w3c98] *Spezifikation von XQL98* [1998].
URL `http://www.w3.org/TandS/QL/QL98/pp/xql.html`

[w3c00] *Extensible Markup Language (XML) 1.0 (Second Edition), W3C Recommendation 6 October 2000* [2000].
URL `http://www.w3.org/TR/REC-xml`

[w3c02a] *W3C XML Development History* [2002].
URL `http://www.w3.org/XML/hist2002`

[w3c02b] *Extensible Markup Language (XML) 1.0 (Zweite Auflage), Deutsche Übersetzung, 20. Januar 2002* [2002].
URL `http://edition-w3c.de/TR/2000/REC-xml-20001006/`

[w3c03] *Process Document* [2003].
URL `http://www.w3.org/Consortium/Process/tr`

[WCO02] L. Wall, T. Christiansen, J. Orwant. *Programmieren mit Perl* (O'Reilly, 2002). ISBN 3-89721-144-0.

[xml98a] *Extensible Markup Language (XML) 1.0, W3C Recommendation 10-February-1998* [1998].
URL `http://www.w3.org/TR/1998/REC-xml-19980210`

[xml98b] *Extensible Markup Language (XML) 1.0, W3C Recommendation 10-February-1998 in deutscher Übersetzung* [1998].
URL `http://www.edition-w3c.de/TR/1998/REC-xml-19980210.html`

[xml99] *XML-in-10-points* [1999].
URL `http://www.w3.org/XML/1999/XML-in-10-points`

[xml01] *XML Query Requirements, W3C Working Draft 15 February 2001* [2001].
URL `http://www.w3.org/TR/2001/WD-xmlquery-req-20010215`

[xml03a] *XML Activity Homepage des W3C Consortiums* [2003].
URL `http://www.w3.org/XML/Activity`

[xml03b] *FastObjects by Poet . XML - Standard sucht passende Datenbank* [2003].
URL `http://www.fastobjects.com/de/fo_de_newsevents_spotlights_020429_body.htm`

[XML03c] *XML-Query-Language* [2003].
URL `http://www.cuesoft.com/xqlspec.htm`

[xml03d] *XML Query Homepage des W3C* [2003].
URL `http://www.w3.org/XML/Query`

[xpa03] *XPath* [2003].
URL `http://www.w3.org/TR/xpath`

[XQL99] *XQL '99 Proposal* [1999].
URL `http://www.ibiblio.org/xql/xql-proposal.html`

[xsl03] *XSLT* [2003].
URL `http://www.w3.org/TR/xslt`

[You00] M. Young. *XML Schritt für Schritt* (Microsoft Press, 2000). ISBN 3-86083-765-7.

[Zie01] C. Ziegler. *XML und Datenbanken: Einblick in Tamino*. *iX*, Band 6 [2001].

B

Anhang A

Evaluierungsdaten

A.1 Benchmark Perl-Modul XML-XQL

Alle Tests wurden auf einem Linux-Rechner mit 750MHz CPU-Leistung und 314 Megabyte Hauptspeicher durchgeführt.

Teststellung

Für die Untersuchung der Performance von *xql.pl* wurden vier Benchmarks mit jeweils unterschiedlichen XML-Dokumenten durchgeführt. Die XML-Dokumente sind so gewählt worden, daß sie jeweils einen bestimmten Grenzfall beschreiben. Die erste Messreihe untersucht den Zeitaufwand beim Durchsuchen einer großen Anzahl kurzer XML-Dateien. In der zweiten Messreihe wurde das Verhalten bei längeren, vielen Dateien geprüft. Die dritte Messreihe hatte zum Ziel, den Zeitaufwand bei der Suche in einem großen XML-Dokument mit vielen Tags zu untersuchen. Bei der vierten Messreihe wurde untersucht, ob die Größe des Inhalts der Tags Einfluss auf die Suchzeit hat.

Evaluierungsdaten

Versuch 1 Suche in 10000 XML-Files, mit jeweils zehn Knoten, einer Verschachtelungstiefe von vier, 15 Zeilen und einer Größe von 433 Bytes.

Versuch 2 Suche in 1000 XML-Files, mit jeweils 54 Knoten, einer Verschachtelungstiefe von vier, 84 Zeilen und einer Größe von 4 Kilobyte.

Versuch 3 Suche in einem XML-File, mit 540 Knoten, einer Verschachtelungstiefe von fünf, 840 Zeilen und einer Größe von 46 Kilobyte. Für diesen Test wurde der Inhalt der Files aus Versuch 2 in ein großes File kopiert.

Versuch 4 Suche einem XML-File, mit zwei Knoten, einer Verschachtelungstiefe von zwei, 1000 Zeilen und einer Größe von 55 Kilobyte.

Files	Knoten	Bedingungen	Treffer	Antwortzeit
10000	10	1	1	69s
		2	700	70s
		3	700	72s
		4	1	72s
1000	54	0	100	24s
		2	40	25s
		4	1	25s
		6	100	25s
1	540	0	1	2s
		1	20	2s
		4	1	2s
1	2	0	1	1s

Tabelle A.1: Messwerte Perl-Modul XML-XQL

Bewertung der Ergebnisse

Das Ergebnis der Benchmarks hinsichtlich der Performance ist, dass das Suchen in einer grossen Datei, die mehrere Datensätze enthält, weniger Zeit in Anspruch nimmt als in vielen kleinen XML-Dateien. So sind die Suchzeiten in den Versuchen 3 und 4 deutlich geringer als in den anderen beiden. Allerdings ist die Verwaltung einer so großen Datei eher unpraktisch.

Die Ergebnisse zeigen ausserdem , daß es keine Rolle spielt, ob es eine hohe oder niedrige Anzahl von Treffern gibt. Auch bei der Benutzung von unterschiedlich vielen logischen Verknüpfungen konnten keine wesentlichen Unterschiede in den Zeiten festgestellt werden. Jedoch sind die Zeiten für 1000 und mehr Files sehr hoch.

A.2 Benchmark ASIX

A.2.1 Messwerte der Testreihe 1

Der Zeitaufwand ohne Index

XML-Verschachtelungstiefe	Anzahl XML-Files	Anzahl Bedingungen in XQL-Query	Zeit in ms
1	20	1	1337
		2	1353
		3	1363
		4	1376
	200	1	2899
		2	2902
		3	3022
		4	3061
	2000	1	17497
		2	17760
		3	18787
		4	18562
2	20	1	1523
		2	1413
		3	1400
		4	1404
	200	1	3559
		2	3517
		3	3427
		4	3391
	2000	1	24361
		2	22138
		3	22790
		4	22288
3	20	1	1477
		2	1477
		3	1596
		4	1604
	200	1	4048
		2	3936
		3	4219
		4	4331
	2000	1	29996
		2	28184
		3	31631

Fortsetzung auf der nächsten Seite

XML-Verschachtelungstiefe	Anzahl XML-Files	Anzahl Bedingungen in XQL-Query	Zeit in ms
		4	32110
4	20	1	1503
		2	1558
		3	1531
		4	1509
	200	1	4297
		2	4225
		3	4435
		4	4506
	2000	1	31329
		2	28605
		3	30308
		4	29634

Tabelle A.2: Messwerte ASIX ohne Index

A.2.2 Messwerte der Testreihe 2

Der Zeitaufwand mit Index

XML-Verschachtelungstiefe	Anzahl XML-Files	Anzahl Bedingungen in XQL-Query	Zeit in ms
1	20	1	214
		2	214
		3	216
		4	220
	200	1	259
		2	272
		3	282
		4	314
	2000	1	682
		2	816
		3	948

Fortsetzung auf der nächsten Seite

XML-Verschachtelungstiefe	Anzahl XML-Files	Anzahl Bedingungen in XQL-Query	Zeit in ms
		4	1210
2	20	1	216
		2	214
		3	241
		4	228
	200	1	257
		2	279
		3	294
		4	318
	2000	1	763
		2	821
		3	959
		4	1173
3	20	1	214
		2	222
		3	220
		4	225
	200	1	258
		2	275
		3	303
		4	341
	2000	1	776
		2	846
		3	1231
		4	1549
4	20	1	216
		2	225
		3	229
		4	228

Fortsetzung auf der nächsten Seite

XML-Verschachtelungstiefe	Anzahl XML-Files	Anzahl Bedingungen in XQL-Query	Zeit in ms
	200	1	260
		2	274
		3	327
		4	321
	2000	1	704
		2	820
		3	1086
		4	1185

Tabelle A.3: Messwerte ASIX mit Index

A.2.3 Zeitersparnis und Verbesserungsfaktor

XML-Tagtiefe	XML-Files	Bedingungen in XQL-Query	Zeitverbesserung in ms	Verbesserungsfaktor
1	20	1	1123	6,25
		2	1139	6,32
		3	1147	6,31
		4	1156	6,25
	200	1	2640	11,19
		2	2630	10,67
		3	2740	10,72
		4	2747	9,75
	2000	1	17078	26,04
		2	16681	21,44
		3	17839	19,82
		4	7352	15,34

Fortsetzung auf der nächsten Seite

XML-Tag-tiefe	XML-Files	Bedingungen in XQL-Query	Zeitverbes-serung in ms	Verbesser-ungsfaktor
2	20	1	1307	7,05
		2	1199	6,60
		3	1159	5,81
		4	1176	6,16
	200	1	3302	13,85
		2	3238	12,61
		3	3133	11,66
		4	3073	10,66
	2000	1	23598	31,93
		2	21317	26,96
		3	21831	23,76
		4	21115	19,00
3	20	1	1263	6,90
		2	1255	6,65
		3	1376	7,25
		4	1379	7,13
	200	1	3790	15,69
		2	3661	14,31
		3	3916	13,92
		4	3990	12,70
	2000	1	29220	38,65
		2	27338	33,31
		3	30400	25,70
		4	30561	20,73
4	20	1	1287	6,96
		2	1333	6,92
		3	1302	6,69
		4	1281	6,62

Fortsetzung auf der nächsten Seite

XML-Tag-tiefe	XML-Files	Bedingungen in XQL-Query	Zeitverbes-serung in ms	Verbesser-ungsfaktor
	200	1	4037	16,53
		2	3951	15,42
		3	4108	13,56
		4	4185	14,04
	2000	1	30625	44,50
		2	27785	34,88
		3	29222	27,91
		4	28449	25,01

Tabelle A.4: Antwortzeitverbesserung in ASIX mit Index

Anhang B

Tabellen

B.1 XPath

Achse	Bedeutung
`child`	Kindelemente
`parent`	Direkter Vorgänger
`descendant`	Nachfolger
`ancestor`	Vorgänger
`following`	Alle nach dem gegenwärtigen Knoten im Dokument vorhandenen Knoten
`preceding`	Alle vor dem gegenwärtigen Knoten im Dokument vorhandenen Knoten
`following-sibling`	Geschwister des gegenwärtigen Knotens, die noch folgen
`preceding-sibling`	Geschwister, die vor dem gegenwärtigen Knoten angesiedelt sind
`attribute`	Die Attribute eines Knotens (nur bei Elementen)
`namespace`	Namensraumknoten des gegenwärtigen Knotens (nur bei Elementen)
`self`	Der gegenwärtige Knoten

Fortsetzung auf der nächsten Seite

Achse	**Bedeutung**
`descendant-or-self`	Wie `descendant`, plus der gegenwärtige Knoten
`ancestor-or-self`	Wie `ancestor`, plus der gegenwärtige Knoten

Tabelle B.1: Achsen bei XPath [BM00]

Eigenschaft	**Bedeutung**
`node()`	Liefert *wahr* für alle Knoten.
`text()`	Liefert *wahr* für alle Textknoten.
`*`	Liefert *wahr* für alle Elementknoten.
`comment()`	Liefert *wahr* für alle Kommentarknoten.
`processing-instruction([name])`	Liefert *wahr*, wenn es sich um allgemeine Processing Instructions oder welche mit dem Name `name` handelt.
`qname`	Liefert *wahr*, wenn der aktuelle Knotenname mit `qname` übereinstimmt. Bei der Attributachse steht `qname` für einen Attributnamen. Bei der Namensraumachse steht `qname` für einen Namensraum.

Tabelle B.2: Eigenschaften des Knotentests

Ausdruck	**Abk.**	**Zugriff auf . . .**
`/`		Wurzel des Dokumentenbaumes oder Trennung von Pfadkomponenten
`/descendant-or-self::node()/`	//	Alle Nachkommen des Kontextknotens

Fortsetzung auf der nächsten Seite

Ausdruck	**Abk.**	**Zugriff auf . . .**
`*`		Bezeichnet einen Knoten mit einem beliebigen Namen
`child::`		Die Standardachse
`child::text()`		Alle Textknoten, die Kinder des Kontextknotens sind
`child::node()`		Alle Kindknoten des Kontextknotens
`self::node()`	.	Bezeichnet den aktuellen Knoten, also den Kontextknoten
`parent::node()`	..	Bezeichnet den Vaterknoten
`attribute::*`	@*	Alle Attribute des Kontextknotens
`descendant::number`		Alle `number`-Nachfolger des Kontextknotens
`/descendant::number`		Alle `number`-Elemente
`child::number [position()=1]`		Erstes `number`-Element des Kontextknotens
`child::number [position()=last()]`		Letztes `number`-Element des Kontextknotens
`child::call_numbers /descendat::number`		Alle `number`-Elemente der `call_numbers`-Kindelemente

Tabelle B.3: Beispiele von Ausdrücken in XPath

B.2 Quilt

Operator	Bedeutung
.	Aktueller Knoten
/	Direkte Nachfolger (Kinder) des aktuellen Knoten
//	Alle Nachfolger des aktuellen Knotens, entspricht der Hülle über /
@	Bezeichnet ein Attribut
[]	Die Klammern umschließen einen boolschen Ausdruck, der als Prädikat für einen Navigationsschritt dient.
[n]	Sofern ein Prädikat aus einer ganzen Zahl besteht, beschreibt es die Ordinalposition eines Elements innerhalb einer Liste von Elementen.
document(string)	Bestimmt die Wurzel eines XML-Baums string.
->	Der Dereferenzierungsoperator.

Tabelle B.4: Pfadoperatoren in Quilt [Sta02]

B.3 Regular Expressions

Symbol	Bedeutung
Allg. Metazeichen	
\ ...	Nächstes nicht-alphanumerisches Zeichen wird kein Metazeichen und umgekehrt (vielleicht)
(... | ...)	Alternative Erkennt das eine oder das andere
(...)	Gruppierung
[...]	Zeichenklasse Erkennt ein Zeichen aus einer Menge
[^ ...]	Alle Zeichen, die nicht in der Menge sind
^	Wahr am Anfang des Strings
.	Erkennt jedes Zeichen außer Newline
$	Wahr am Ende des Strings
Alphanumerische Metasymbole	
\0	Paßt auf das Nullzeichen
\NNN	Paßt auf ein oktal angegebenes Zeichen
\n	Paßt auf den n-ten vorher festgehaltenen String
\a	Paßt auf das Signalzeichen
\A	Wahr am Anfang des Strings
\b	Wahr, wenn Wortgrenze
\B	Wahr, wenn nicht Wortgrenze
\cX	Paßt auf das Steuerzeichen Strg+X
\C	Paßt auf ein Byte
\d	Paßt auf eine Ziffer
\D	Paßt auf ein Nichtziffer-Zeichen
\e	Paßt auf das Escape-Zeichen
\f	Paßt auf einen Seitenvorschub
\G	Wahr an Match-Ende-Position der

Fortsetzung auf der nächsten Seite

Symbol	**Bedeutung**
	Vorangegangenen m//g-Operation
\l	Wandelt das nachfolgende Zeichen
	in einen Kleinbuchstaben
\L	Wandelt nachfolgende Zeichen bis zum
	nächsten \E in Kleinbuchstaben
\n	Paßt auf das Newline-Zeichen
\N{NAME}	Paßt auf das genannte Zeichen
	\N{greek:Sigma}
\p{EIGENSCHAFT}	Paßt auf jedes Zeichen mit der EIGENSCHAFT
\P{EIGENSCHAFT}	Paßt auf jedes Zeichen ohne die EIGEN-
	SCHAFT
\Q	Quoting von Metazeichen bis \E
\r	Paßt auf Wagenrücklauf
\s	Paßt auf das Whitespace-Zeichen
\S	Paßt auf Nicht-Whitespace-Zeichen
\t	Paßt auf das Tabulator-Zeichen
\u	Nächstes Zeichen in Titelschreibweise
\U	Wandelt nachfolgende Zeichen bis zum
	nächsten \E in Großbuchstaben
\w	Paßt auf ein Wort-Zeichen
\W	Paßt auf ein Nicht-Wort-Zeichen
\x{abcd}	Paßt auf ein hexadezimal engegebenes Zeichen
\X	Paßt auf Zeichenkombination in Unicode
\z	Wahr am Ende des Strings
Quantoren beziehen sich auf das unmittelbar davor stehende Zeichen	
*	Nullmal oder mehrfach
+	Einmal oder mehrfach
?	Nullmal oder einmal

Fortsetzung auf der nächsten Seite

Symbol	Bedeutung
{ANZAHL}	Genau ANZAHL-mal
{MIN,}	MIN-mal bis unendlich
{MIN,MAX}	Mindestens MIN-mal
	höchsten MAX-mal

Tabelle B.5: Reguläre Ausdrücke in Perl

Std	Gen.	Bezeichnung	Intp	Beispiel
' '	q{}	Wörtlich	Nein	'$100' q!$100!
" "	qq{}	Interpoliert	Ja	"$var" qq#$var#
` `	qx{}	Backticks	Ja	`who -r` qx{$unx_cmd}
	qw{}	Wortliste	Nein	qw(a b c) ('a','b','c')
//	m{}	Pattern Matching	Ja	/$abc/ m#/usr/include#
	s{}{}	Suchen und Ersetzen	Ja	s!$abc!xyz!g
	tr{}{}	Translation	Nein	tr/a-z/A-Z/

Tabelle B.6: Quoting-Operatoren

B.4 Das POD-Format

Anweisung	Syntax	Bedeutung
=head1	=head1 *Überschrift*	Hauptüberschrift
=head2	=head2 *Überschrift*	Unterüberschrift
=over	=over *Anzahl Spalten*	nach rechts einrücken
=back	=back	Ende der Einrückung
=item	=item *Listenzeichen*	Eintrag einer Liste =item 1. für Zahlen =item * für Bullets
=pod	=pod	Beginn der POD-Sektion
=cut	=cut	Ende von POD

Tabelle B.7: POD-Kommandos aus [Haj98]

Format	Syntax	Bedeutung
Italic	I< *text* >	Kursivschrift für Variablenhervothebung
Bold	B< *text* >	Fettschrift für Switches und Programme
Space	S< *text* >	Text mit Leerzeichen
Code	C< *code* >	Code, Verbatim
Link	L< *name* > L< *name/ident* > L< *name/"sec"* > L< [/]*"sec"* >	Querverweis zu anderer Manpage Querverweis zu Item in Manpage Sektion in Manpage Sektion hier
File	F< *file* >	für Dateinamen
Index	X< *index* >	ein Indexeintrag

Tabelle B.8: POD-Formatierungsanweisungen aus [Haj98]

Anhang C

Quellen

C.1 Die Konfigurationsdatei

Listing C.1: asix.conf

```
# Configuration of ASIX

# Path of any variable files incl. index files
VAR_PATH=/var/asix/

# Path of required perl modules
LIB_PATH=/usr/share/perl5/ASIX/lib/

# Values to set the triggers to build an index
# if one value is true the index will be build automatically

# max searchtime in seconds
MAX_SEARCHTIME=3

# last access in days
MAX_LASTACCESS=60

# min number of query calls
MIN_QUERYCALLS=3

# name of logfile
LOG=logfile.lg
```

C.2 Indextabellen

`VAR_PATH=/var/asix/`	Verzeichnis, in dem zur Laufzeit erzeugte Dateien erstellt werden
`LIB_PATH=/usr/share/perl5/ASIX/lib/`	Verzeichnis der Bibliotheken
`MAX_SEARCHTIME=3`	Wird der Wert der maximalen Suchzeit von einer Query überschritten, wird ein Index erzeugt.
`MAX_LASTACCESS=60`	Wird der Wert des letzten Aufrufs von einer Query überschritten, wird der der Query Index gelöscht.
`MIN_QUERYCALLS=3`	Wird eine Query öfter aufgerufen als dieser Wert, wird ein Index erzeugt.
`LOG=log4xqlquery.log`	Name der Logdatei, in der die Suchanfragen gespeichert werden.

Tabelle C.1: Erklärung zum Listing von asix.conf

Listing C.2: group__assistant__name__lastname.group__assistant__private__age.idx

```
Filename group__assistant__name__lastname group__assistant__private__age
__home__hotzky__asix__files__test1.xml Smith      26
__home__hotzky__asix__files__test2.xml Hackleton  23
__home__hotzky__asix__files__test3.xml Regnault   18
__home__hotzky__asix__files__test4.xml Rutherford 20
__home__hotzky__asix__files__test5.xml Murphy     42
__home__hotzky__asix__files__test6.xml Scott      33
__home__hotzky__asix__files__test7.xml Collins    29
__home__hotzky__asix__files__test8.xml Cook       34
```

C.3 Die Logdatei

Listing C.3: logfile.lg

```
Ausgabe                          Bedingung                          Zeitpunkt
            Dauer
group/assistant/name/lastname group/assistant/private/age      166        4
group/assistant/name/lastname group/assistant/name/lastname    166        5
group/assistant/name/lastname group/assistant/private/age      167        3
group/assistant/private_numbers group/assistant/name/lastname 170        4
```

C.4 Die Module

Listing C.4: asix.pl

```
#!/usr/bin/perl
#use strict;

=head1 NAME

ASIX - A Perl Pogram that searchs in XML-Files

=head1 USAGE

Searching in xml-files:

$ B<ASIX.pl -s "xql-query" "xml-pfad">

$ C<ASIX.pl -s "/bookstore/book/price" "xml/*.xml">

Buiding up the index:

$ B<ASIX.pl -i "xml-pfad">

$ C<ASIX.pl -i "xml/">

Deleting one xml-file

$ B<ASIX.pl -d "xml-datei">

$ C<ASIX.pl -d "xml/datei1.xml">

=head1 DESCRIPTION

The program ASIX can be run in three different modes.

Searching Mode: Start with the option C<-s> to search in XML files.
The first parameter is a XQL query. The second is a path with XML files in
              it
or a single XML file.

Index Mode: Start with the option C<-i> to build the index. The first
parameter is the path where the index files shell built. The second
             parameter
is the path where the XML files are.

Deletion Mode: Start with the option C<-d> to delete a XML file. The first
parameter is the path where the index files are. The second is a single
XML file that shell be deleted.
```

```
=head1 INSTALLATION

Run F<setup> to install all required files in super user mode.

To use the search functions the package F<XML-XQL> at
http://search.cpan.org/search?dist=XML-XQL must be installed.
The script F<xql.pl> in F<XML-XQL-0.xx/bin> must be copied to F</usr/bin>

=head1 METHODS

B<startxql()>

=over 1

Starts functions of the searching algorithm.

=back

B<startidx()>

=over 1

Starts function to build the index.

=back

B<notify()>

=over 1

Starts functions to update the index when a xml-file was modified.

=back

B<del()>

=over 1

Starts functions to delete a file and update the index.

=back

%global is a hash with global value read from /etc/asix.conf and command
            line
parameters.
```

```
=head1 AUTHOR

Colin Hotzky, c@hotzky.de

=head1 COPYRIGHT

Copyright 2003, Colin Hotzky.

=cut

use lib '/usr/share/perl5/ASIX/lib/';
use update;
use xqlsuche;
use indexer;

sub export_value {
    $ENV{ $_[0] } = ${ $_[0] };
    *{ $_[0] } = \( $ENV{ $_[0] } );
}

print "**********************************************\n";
print "***                ASIX                    ***\n";
print "***          Automatic Search in XML       ***\n";
print "***       Copyright 2003 by Colin Hotzky   ***\n";
print "**********************************************\n\n";

open( INIT, "/etc/asix.conf" );
my @init = <INIT>;
close(INIT);

my %global;
for my $el (@init) {
    chomp($el);
    my ( $key, $value );
    if ( $el =~ /VAR_PATH/ ) {
        ( $key, $value ) = split ( /=/, $el );
        $VAR_PATH = $value;
        export_value VAR_PATH;
    }
    elsif ( $el =~ /LIB_PATH/ ) {
        ( $key, $value ) = split ( /=/, $el );
        $LIB_PATH = $value;
        export_value LIB_PATH;
    }
    elsif ( $el =~ /MAX_SEARCHTIME/ ) {
        ( $key, $value ) = split ( /=/, $el );
```

```
        $MAX_SEARCHTIME = $value;
        export_value MAX_SEARCHTIME;
    }
    elsif ( $el =~ /MAX_LASTACCESS/ ) {
        ( $key, $value ) = split ( /=/, $el );
        $MAX_LASTACCESS = $value;
        export_value MAX_LASTACCESS;
    }
    elsif ( $el =~ /MIN_QUERYCALLS/ ) {
        ( $key, $value ) = split ( /=/, $el );
        $MIN_QUERYCALLS = $value;
        export_value MIN_QUERYCALLS;
    }
    elsif ( $el =~ /LOG/ ) {
        ( $key, $value ) = split ( /=/, $el );
        $LOG = $ENV{VAR_PATH} . $value;
        export_value LOG;
    }
}

# ARGV[0] = -s
# ARGV[1] = xql-query --- /bookstore/book/price
# ARGV[2] = xml-pfad  --- xml/*.xml

if ( $ARGV[0] eq "-s" ) {
    $XQL_QUERY = $ARGV[1];
    export_value XQL_QUERY;
    $XML_PATH = $ARGV[2];
    export_value XML_PATH;
    $BIN_PATH = "/usr/bin";
    export_value BIN_PATH;
    $MODE = "search";
    export_value MODE;
    startxql();
}

# ARGV[0] = -d
# ARGV[2] = xml-datei    --- xml/datei1.xml

elsif ( $ARGV[0] eq "-d" ) {
    $XML_FILE = $ARGV[1];
    export_value XML_FILE;
    del();
}

# ARGV[0] = -i
```

```
# ARGV[1] = index-pfad   --- index/
# ARGV[2] = xml-pfad          --- xml/

elsif ( $ARGV[0] eq "-i" ) {
    $XML_PATH = $ARGV[1];
    export_value XML_PATH;
    startidx();
}

# ARGV[0] = -u
# ARGV[1] = idx-pfad          --- index/
# ARGV[2] = xml-pfad          --- xml/

elsif ( $ARGV[0] eq "-u" ) {
    $XML_PATH = $ARGV[1];
    export_value XML_PATH;
    notify();
}

elsif ( $ARGV[0] eq "--help" ) {
    print <<EOF;
Um in xml-files zu suchen:
asix.pl -s "xql-query" "xml-pfad" ["xml-datei"]

Um ein xml-file aus der Datenbank zu loeschen:
asix.pl -d "index-pfad" "xml-pfad" "xml-datei"

Um die index-files zu erzeugen:
asix.pl -i "index-pfad" "xml-pfad"
EOF
}
else { print "\nType\nasix.pl␣--help\nfor␣help.\n" }

print "\nHURRA␣-␣Programm␣erfolgreich␣beendet!\n";
```

Listing C.5: mynotify.pm

```
#!/bin/bash
# mynotify xmlpath

xmlpath=$1
ls -l $1*.xml > /var/asix/ls.dat
echo 'Starting dnotify...'
dnotify -M $xmlpath -e perl asix.pl -u $xmlpath
```

Listing C.6: xqlsuche.pm

```
#!/usr/bin/perl
package xqlsuche;    #ohne .pm
use strict;

use vars qw(@ISA @EXPORT @EXPORT_OK %EXPORT_TAGS $VERSION);
use Exporter;

$VERSION = 1.0 ;
@ISA = qw(Exporter);

@EXPORT = qw (startxql);

# ------------------------------------------------------------------------

# @author: Colin Hotzky
# @date: Februar-Mai 2003
# @copyright: 2003 by Colin Hotzky
# @contact: c@hotzky.de
# @description: Programm das in XML-Dateien sucht
# ------------------------------------------------------------------------

# parameter:

use lib "./";
use startProgram;
use func;
use xql2sql;
use strict;

sub pm {
    print "\n---␣---";
    my $feld = shift (@_);
    for (@_) { print "\n$feld:␣$_"; }
}

sub startxql {
    print "\n␣*********************************************\n";
    print "␣***␣XQL2SQL-Converter␣V1.3␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣***\n";
    print "␣***␣Copyright␣2003␣by␣Colin␣Hotzky␣␣␣␣␣␣␣␣***\n";
    print "␣*********************************************\n\n";

    my $condition = "no";

    my %hash;
    my $searchtime;
```

```
45      # test, ob es eine bedingung gibt
        $condition = "yes" if ( $ENV{XQL_QUERY} =~ /\[/ );

        # just pick the paths out
        if ( $condition eq "yes" ) {
50          my (@words) = split ( /\s/, $ENV{XQL_QUERY} );

            my @paths;

            for (@words) {
55              push ( @paths, $_ ) if ( ( $_ =~ /\// ) and !( $_ =~ /'/␣)␣);
            }

            my $firstel = shift (@paths);
            my @el      = split ( /\[/, $firstel );
60
            if ( $paths[0] =~ /\.$/ ) {
                unshift ( @paths, $el[0] );
            }
            else {
65              unshift ( @paths, $el[1] );
                unshift ( @paths, $el[0] );
            }

            for ( my $i = 0 ; $i < scalar @paths ; $i++ ) {
70              substr( $paths[$i], 0, 1 ) = "" unless ( $paths[$i] =~ /^\// )
                            ;
            }

            # doppelte Einträge zählen und speichern
            my @pseudo;
75
            my $pseudoref = \@pseudo;
            my $pathsref  = \@paths;

            ( $pathsref, $pseudoref ) = uniq( \@paths, \@pseudo );
80
            # pfade => indexname
            my $index = build_index_file_name(@paths);

            # 1 = ex, 0 = ex. nicht
85          if ( exist_index_file($index) == 0 ) {
                my $strpath = "";
                for my $el (@paths) { $strpath .= $el . "␣"; }
```

```
            substr( $strpath, -1, 1 ) = "";

            # XQL-Suche
            $hash{paths}     = $strpath;
            $hash{programm}  = "xql.pl";
            $hash{xqlquery}  = "\"" . $ENV{XQL_QUERY} . "\"";
            $hash{database}  = $ENV{XML_PATH};
            $hash{result}    = "yes";
            $hash{condition} = "yes";
            $hash{mode}      = "search";

            $searchtime = startProgram(%hash);
        }
        else {
            my %sprite = xql2sql();

            # SQL-Suche
            $hash{programm}  = "sprite.pl";
            $hash{database}  = $ENV{VAR_PATH} . $index;
            $hash{select}    = $sprite{select};
            $hash{condition} = $sprite{condition};

            # Format der Parameter anpassen
            $hash{select}    =~ s/^\///;
            $hash{select}    =~ s/\//__/g;
            $hash{condition} =~ s/^__//;

            $searchtime = startProgram(%hash);
        }
    }
    else {
        my $indexfile = $ENV{XQL_QUERY} . ".idx";
        if ( exist_index_file($indexfile) == 0 ) {

            # XQL-Suche
            $hash{paths}     = $ENV{XQL_QUERY};
            $hash{programm}  = "xql.pl";
            $hash{xqlquery}  = "\"" . $ENV{XQL_QUERY} . "\"";
            $hash{database}  = $ENV{XML_PATH};
            $hash{result}    = "yes";
            $hash{condition} = "no";
            $hash{mode}      = "search";

            $searchtime = startProgram(%hash);
        }
        else {
```

```
            my %sprite = xql2sql();

            # SQL-Suche
            $hash{programm} = "sprite.pl";
            $hash{database} = $ENV{VAR_PATH} . $indexfile;
            $hash{select}   = $sprite{select};

            # Format der Parameter anpassen
            $hash{select} =~ s/^\///;
            $hash{select} =~ s/\//__/g;

            $searchtime = startProgram(%hash);
        }
    }

}

1;
```

Listing C.7: xql2sql.pm

```
#!/usr/bin/perl
package xqlsuche;    #ohne .pm
use strict;

use vars qw(@ISA @EXPORT @EXPORT_OK %EXPORT_TAGS $VERSION);
use Exporter;

$VERSION = 1.0 ;
@ISA = qw(Exporter);

@EXPORT = qw (xql2sql);

# ------------------------------------------------------------------------

# @author: Colin Hotzky
# @date: Mai 2003
# @copyright: 2003 by Colin Hotzky
# @contact: c@hotzky.de
# @description: Programm das XQL-Syntax in SQL-Syntax umwandelt
# ------------------------------------------------------------------------

# parameter:

use lib "./";
use startProgram;
use indexer;
use func;
use strict;

sub pm {
    print "\n---␣---";
    my $feld = shift (@_);
    for (@_) { print "\n$feld:␣$_"; }
}

sub xql2sql {
    my $xqlquery = $ENV{XQL_QUERY};

    # Query aufsplitten
    my (@words) = split ( /\s/, $xqlquery );

    my @h1;

    # ausgabe von bedingung trennen
    (@h1) = split ( /\[/, shift (@words) );
```

```
    my %sql;
    $sql{select} = $h1[0];

    # . durch ausgabe ersetzen
    if ( $h1[1] =~ /\.$/ ) {
        $h1[1] =~ s/\./$h1[0]/;
    }

    # ersten bedingungspfad wieder @words hinzufuegen
    unshift ( @words, $h1[1] );

    # query fuer sql anpassen
    for ( my $i = 0 ; $i < scalar @words ; $i++ ) {
        if ( $words[$i] =~ /(and|or|not)/i ) {
            $words[$i] =~ s/\$//g;
        }
        $words[$i] =~ s/^ilt$/</g;
        $words[$i] =~ s/^ile$/<=/g;
        $words[$i] =~ s/^igt$/>/g;
        $words[$i] =~ s/^ige$/>=/g;
        $words[$i] =~ s/^ie$/=/g;

        if ( ( $words[$i] =~ /^\// ) || ( $words[$i] =~ /^\(\// ) ) {
            $words[$i] =~ s/^\///;
            $words[$i] =~ s/^\(\//\(/;
            $words[$i] =~ s/\//__/g;
        }
        if ( $words[$i] =~ /^'/␣)␣{
            $words[$i] =~ s/'//g;
        }

    }

    # letzes ] loeschen
    substr( $words[-1], -1, 1 ) = "" if ( $words[-1] =~ /\]$/ );

    my $sqlausgabe = "";
    for (@words) {
        $sqlausgabe .= $_ . "␣";
    }
    substr( $sqlausgabe, -1, 1 ) = "";
    $sql{condition} = $sqlausgabe;
    return %sql;
}
1;
```

Listing C.8: startProgram.pm

```
#!/usr/bin/perl
package startProgram;    #ohne .pm
use strict;

use vars qw(@ISA @EXPORT @EXPORT_OK %EXPORT_TAGS $VERSION);
use Exporter;
use xml2txt;
use func;

$VERSION = 0.1 ;
@ISA = qw(Exporter);

@EXPORT = qw (startProgram);

# -----------------------------------------
# ---- Aufruf eines externen Programms  ----
# -----------------------------------------

sub startProgram          # hash: programm, database, select, where
{
    my %hash = @_;

    my $start = time();
    my $srg   = "";
    my $Name;

    if ( $hash{programm} eq "sprite.pl" ) {
        $srg .= $hash{programm} . " ";
        $srg .= $hash{database} . " \"";
        $srg .= $hash{select} . "\" \"";
        $srg .= $hash{condition} . "\"";
        $Name = "Sprite";
    }
    elsif ( $hash{programm} eq "xql.pl" ) {
        $srg .= $hash{programm} . " ";
        $srg .= $hash{xqlquery} . " ";
        $srg .= "-H %s ";

        $srg .= $ENV{XML_PATH};

        $srg .= "*.xml"  unless ( $srg =~ /(\*\.xml)$/ );
        $srg .= "/*.xml" unless ( $srg =~ /(\/\*\.xml)$/ );

        $srg .= "> $ENV{VAR_PATH}result";
        $Name = "xql";
```

```
    }

    print "\n\nAufruf:\n\$ $srg\n";
    $srg .= " 2> /dev/null";
    my @args = ($srg);

    my $rc = 0xffff & system @args;
    select STDERR;

    if ( $rc == 0 ) {
    }
    elsif ( $rc == 0xff00 ) { print "\nAufruf ($srg) fehlgeschlagen: $!";
                die; }
    elsif ( $rc > 0x80 ) {
        $rc >>= 8;
        if ( $rc != 1 ) {
            print
              "\nProgramm ($Name) mit Exit-Status ungleich null beendet:
                          $rc";
            die;
        }
        else {
            print "\n*****************************************";
            print "\n*** Die Suche erzielte keine Treffer !!! ***";
            print "\n*****************************************\n";
        }
    }
    else {

        if ( $rc & 0x80 ) {
            $rc &= ~0x80;

        }

    }
    my $ok = ( $rc != 0 );
    select STDOUT;
    $hash{searchtime} = time() - $start;
    if ( $hash{programm} eq "xql.pl" ) {

        # ---- ---- ---- ---- ---- ---- ---- ---- ---- ----
        xml2txt(%hash);

        # ---- ---- ---- ---- ---- ---- ---- ---- ---- ----
        if ( ( $hash{condition} eq "yes" ) && ( $hash{mode} eq "search" )
                    ) {
```

```
            func::logging(%hash);

        }
    }

    return 1;
}
```

Listing C.9: xml2txt.pm

```
#!/usr/bin/perl
# ------------------------------------------------------------------------

# @author: Colin Hotzky
# @date: April-August 2002
# @copyright: 2002 by Colin Hotzky
# @contact: c@hotzky.de
# @description: Programm, das XQL-Ergebnisse in XML-Form in txt speichert
# ------------------------------------------------------------------------

package xml2txt;    #ohne .pm
use strict;

use vars qw(@ISA @EXPORT @EXPORT_OK %EXPORT_TAGS $VERSION);
use Exporter;

use lib "./";

$VERSION = 1.0 ;
@ISA = qw(Exporter);

@EXPORT = qw (xml2txt);

sub xml2txt {

    # [0]: Query
    # [1]: firstsearch xml-Datei von erster Suche
    # [2]: xqlsearch von erster Suche
    my %hash        = @_;
    my $query       = $hash{xqlquery};
    my $dateiname   = $ENV{VAR_PATH} . "result";
    my $XMLHeader1  = $hash{ausgabe};

    print "\n\n***␣Ergebnis␣der␣Suche␣***\n\n" if ( $hash{result} eq "yes"
                );
    open( READ, "$dateiname" );
    my @ilesen = <READ>;
    close(READ);

    #Variablen der Schleife
    my @file;
    my @wert;
    my $header;
    my $j;
    my $k;
```

```
    my $l;

    # Schleifen für READ
    for ( my $i = 0, $k = -1 ; $i < ( scalar @ilesen ) ; $i++ ) {
        if ( $ilesen[$i] =~ /\.xml/ ) {
            $j = 0;
            $k++;
            ( $file[$k], $wert[$k][$j] ) = split ( /</, $ilesen[$i] );
            chomp( $wert[$k][$j] );
            substr( $wert[$k][$j], 0, 0 ) = "<";
            ( $header, $wert[$k][$j] ) = split ( />/, $wert[$k][$j] );
            $j++;
        }
        else {
            ( $header, $wert[$k][$j] ) = split ( /</, $ilesen[$i] );
            chomp( $wert[$k][$j] );
            substr( $wert[$k][$j], 0, 0 ) = "<";
            ( $header, $wert[$k][$j] ) = split ( />/, $wert[$k][$j] );
            $j++;

        }
        ( $l, $header ) = split ( /</, $header );
    }

    open( WRITE, ">$ENV{VAR_PATH}xml.txt" );

    #-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

    my $max = 0;

    for my $i ( 0 .. $#wert ) {
        for my $j ( 0 .. $#{ $wert[$i] } ) {
            $max = $j if ( $max < $j );
        }
    }

    $header =~ s/\//__/g;
    substr( $XMLHeader1, 0, 1 ) = "" if ( $XMLHeader1 =~ /^\// );    #/

    $XMLHeader1 =~ s/\//__/g;

    for my $i ( 0 .. $max ) {
        print WRITE "FileName\t$XMLHeader1";
        if ( $hash{result} eq "yes" ) {
            my $x = $XMLHeader1;
            $x =~ s/__/\//g;
```

```
            print STDOUT "FileName\t$x\n";
        }
    }
    print WRITE "\tQuery\n";

    for my $i ( 0 .. $#wert ) {
        $file[$i] =~ s/\//__/g;
        $query =~ s/\//__/g;
        print WRITE $file[$i];
        if ( $hash{result} eq "yes" ) {
            my $x = $file[$i];
            $x =~ s/__/\//g;
            print STDOUT $x;
        }
        my $j = 0;

        #for $j (0 .. $#{$wert[$i]})
        {
            print WRITE "\t$wert[$i][$j]";
            print STDOUT "\t$wert[$i][$j]\n" if ( $hash{result} eq "yes" )
                        ;
            print WRITE "\t$query";
        }
        print WRITE "\n";
    }

    close(WRITE);

    #-+-+--+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
    select STDOUT;
    print "\n\n*************************\n\n" if ( $hash{result} eq "yes"
                );
}
1;
```

Listing C.10: indexer.pm

```
#!/usr/bin/perl -w
package indexer;    #ohne .pm
use strict;

use vars qw(@ISA @EXPORT @EXPORT_OK %EXPORT_TAGS $VERSION);
use Exporter;

$VERSION = 1.0 ;
@ISA = qw(Exporter);

@EXPORT = qw (startidx);

# ------------------------------------------------------------------------

# @author: Colin Hotzky
# @date: March 2002 - Mai 2003
# @copyright: 2002-2003 by Colin Hotzky
# @contact: c@hotzky.de
# @description: Package, das Index erstellt
# ------------------------------------------------------------------------


use lib "./lib";
use func;
use strict;

sub startidx {
    print "*************************************\n";
    print "***       IndexChecker V3.0        ***\n";
    print "***  Copyright 2003 by Colin Hotzky  ***\n";
    print "*************************************\n\n";

    # Ausgabe <TAB> Bedingung <TAB> Zeitpunkt <TAB> Dauer
    # letzer Aufruf einer Query bestimmen
    # mittlere Suchzeit pro Query
    # laengste Suchzeit einer Query
    # Anzahl Aufrufe einer Query

    open( LOG, $ENV{LOG} ) or die "\nFehler beim öffnen von $ENV{LOG}\n\n"
                ;
    my @logfilelines = <LOG>;
    close(LOG);
    shift (@logfilelines);    # Titelzeile löschen

    my @path;
```

```
    my @date;
    my @last;

    my @loglines = @logfilelines;

    my $number = 0;
    my @path_num_time;

    while ( scalar @logfilelines > 0 ) {
        my $oneline = shift (@logfilelines);
        chomp($oneline);

        my ( $p, $d, $t ) = split ( /\t/, $oneline );

        my $match = "NO";

        # feld [path][day][time]...[time]
        # feld [path][day][time]...[time]

        for ( my $k = 0 ; $k <= $number ; $k++ ) {
            if ( $p eq $path_num_time[$k][0] ) {
                $match = "YES";
                $path_num_time[$k][1]++;
                $path_num_time[$k][2] .= "␣" . $t;
            }
        }
        if ( $match ne "YES" ) {
            $path_num_time[$number][0] = $p;
            $path_num_time[$number][1] = 1;
            $path_num_time[$number][2] = $t;
            $number++;
        }

    }

    print "\n" . $number . "\n";
    for my $i ( 0 .. $#path_num_time ) {
        print "xxx␣[@{$path_num_time[$i]}],\n";
    }

    my $mw = 0;
    my @zaehler;

    # mittelwert der suchzeiten
    for ( my $k = 0 ; $k < $number ; $k++ ) {
        (@zaehler) = split ( /\s/, $path_num_time[$k][2] );
```

```
        print "\n" . scalar @zaehler;
        my $nenner = scalar @zaehler;
        my $summe  = 0;

        for my $z (@zaehler) { $summe += $z; }

        $mw = $summe / $nenner;
        $path_num_time[$k][2] = $mw;
    }

    for my $i ( 0 .. $#path_num_time ) {
        print "\n[@{$path_num_time[$i]}],\n";
    }

    # -----------------------------------------
    # --- test, ob index erstellt werden muss ---
    # -----------------------------------------

    # path_num_time [path][day][mw]
    for ( my $k = 0 ; $k < $number ; $k++ ) {
        if ( ( $path_num_time[$k][1] > $ENV{MIN_QUERYCALLS} )
            || ( $path_num_time[$k][2] > $ENV{MAX_SEARCHTIME} ) )
        {

            # index erstellen
            my (@paths) = split ( /\s/, $path_num_time[$k][0] );

            # pfade => indexname
            my $index = build_index_file_name(@paths);
            print "\nindex: $index";

            # ex. index?
            if ( exist_index_file($index) == 0 ) {

                # index mit xql erzeugen
                print "\nIndexfile $index erzeugen...";
                indexsuche( $index, $ENV{XML_PATH} );
            }
        }
    }

}
1;
```

Listing C.11: update.pm

```
#!/usr/bin/perl
# ----------------------------------------------------------------------

# @author: Colin Hotzky
# @date: Januar 2003
# @copyright: 2003 by Colin Hotzky
# @contact: c@hotzky.de
# @description: Programm, das die Indexdateien updated
# ----------------------------------------------------------------------

package update;    #ohne .pm
use strict;

use vars qw(@ISA @EXPORT @EXPORT_OK %EXPORT_TAGS $VERSION);
use Exporter;

use lib "./";
use func;

$VERSION = 1.0 ;
@ISA = qw(Exporter);

@EXPORT = qw (del notify);

# -------------------------------------------------------
# --- ändert Index, wenn ein XML-files gelöscht wird ---
# -------------------------------------------------------

sub del {

    my $STATUS = "delete";

    # --- bevor eine datei gelöscht wird, wird der name gespeichert
    # --- in allen indexfiles, werden alle einträge mit dieser datei
                gelöscht
    # --- danach wird die datei wirklich gelöscht

    # Datei, die gelöscht werden soll

    $ENV{XML_FILE} =~ s/\//__/g;

    # ------------------------------------------------------------
    check_index( $ENV{XML_FILE}, $STATUS );

    # ------------------------------------------------------------
```

```
    $ENV{XML_FILE} =~ s/__/\//g;

    # Datei löschen
    print "\nloeschen????\n";
    my $x = `rm -i $ENV{XML_FILE}`;
}

# ************************
# *** Hilfsfunktionen ***
# ************************

# ---------------------------------------------------------
# ---                         change                    ---
# --- ändert Index, wenn ein XML-files geändert wurde ---
# ---------------------------------------------------------

sub notify {

    my $found_xml_file = `find $ENV{XML_PATH} -newer $ENV{VAR_PATH}ls.dat
                `;
    return unless ( $found_xml_file =~ /xml/ );
    my $x       = `ls -l $ENV{XML_PATH} > $ENV{VAR_PATH}ls.dat`;
    my $STATUS = "notify";

    my (@splitty) = split ( /\n/, $found_xml_file );

    $found_xml_file = $splitty[-1];

    $found_xml_file =~ s/\//__/g;

    # ------------------------------------------------------------
    check_index( $found_xml_file, $STATUS );

    # ------------------------------------------------------------
    print "\n\n\n";
}

# ---------------------------------------------------------
# ---                    check index                    ---
# --- prüft, ob xml-file in einem Index-file vorkommt ---
# ---------------------------------------------------------

sub check_index {

    # $_[0] = $xml_file
```

```
    # $_[1] = $DEL
    my $xml_file     = $_[0];
    my @index_files = `ls $ENV{VAR_PATH}*.idx`;     # Ergebnisliste von ls
    my $VAR_PATH     = $ENV{VAR_PATH};
    my $STATUS       = $_[1];

    # ***************
    my $EXIST = "no";

    # ***************
    my @add_line;

    # indexfiles durchlaufen
    # löschen, falls aktiviert
    for ( my $i = 0 ; $i < scalar @index_files ; $i++ ) {
        chomp $index_files[$i];
        print "\npruefe: $index_files[$i] ... ";

        open( DAT, "$index_files[$i]" ) or die 'Fehler beim Oeffnen.';

        my @readdat = <DAT>;     # inhalt des index

        # die zeilen, in denen zu löschende Datei vorkommt
        my @test = grep ( /$xml_file/, @readdat );

        # anzahl zeilen von index vor dem löschen
        my $scally = scalar @readdat;

        for ( my $k = 0 ; $k < scalar @readdat ; $k++ ) {
            if ( ( scalar @test > 0 ) && ( $test[0] eq $readdat[$k] ) ) {

                # die datei kommt in einem indexfile vor
                $EXIST = "yes";

                if ( $STATUS eq "delete" || $STATUS eq "notify" ) {

                    # der eintrag wird geloescht
                    splice( @readdat, $k, 1 );
                    print "\nEintrag gelöscht";
                    next;
                }
            }
            print "\n-----------------------------------";
            print "\n$test[0] $readdat[$k] $STATUS $EXIST\n";

        }
```

```
        if ( $STATUS eq "notify" ) {

            # es handelt sich um eine neue datei
            # das ergebnis der xql-suche muss dem index
            # angehöngt werden
            # index_file muss ohne Pfad sein
            my (@trenner) = split ( /\//, $index_files[$i] );    #/
            $xml_file =~ s/__/\//g;

            @add_line = indexsuche( $trenner[-1], $xml_file );

            for (@add_line) { push ( @readdat, $_ ); }
        }

        write_new_index( @readdat, $index_files[$i], $VAR_PATH );

        # Fälle
        # 1. Datei existiert und wird gelöscht
        # 2. Datei existiert und wurde geändert
        # 3. Datei existiert nicht und wird hinzugefügt

        close(DAT);
    }
}

# ------------------------
# --- write new index ---
# ------------------------

sub write_new_index {
    my $PATH       = pop (@_);
    my $index_file = pop (@_);

    # @_ = Indexzeilen minus gelöschte
    my @zeilen = @_;

    # ---- neue Datei schreiben ----
    my $y = $PATH . "test.tst";

    open( DATT, ">$y" ) or die "\n*** Es ist ein Fehler aufgetreten ! ***\n
                ";
    for (@zeilen) { print DATT $_; }
    close DATT;

    # Orginalindex überschreiben
    my $x = `mv $y $index_file`;
```

```
}
1;
```

Listing C.12: func.pm

```
#!/usr/bin/perl
package func;    #ohne .pm
use strict;

use vars qw(@ISA @EXPORT @EXPORT_OK %EXPORT_TAGS $VERSION);
use Exporter;

use lib "./";
use startProgram;

$VERSION = 1.0 ;
@ISA = qw(Exporter);

@EXPORT =
  qw (uniq build_index_file_name exist_index_file average logging
      test_log_file write_log_file test_index_file build_query
                  prepare_index
      indexsuche pm);

# -----------------------------------------
# ---               UNIQ                  ---
# -----------------------------------------
# --- Doppelte Arraywerte löschen --- --- ---
# --- Rückgabe eines Paares (Wert,Anzahl) ---
# -----------------------------------------

sub uniq {

    # @_[0]: $@werteref:/bookstore/book/author/last_name /bookstore/index
                < 5
    # @_[1]: $anzahlref = 0

    my @unique;
    my @anzahl;
    my ( $werteref, $anzahlref ) = @_;

    while ( @$werteref > 0 ) {
        my $el = shift (@$werteref);

        if ( $el =~ /\// )    #/
        {
            my $i = -1;
            my $k;

            # auf dopppelte Einträge prüfen
```

```
            for ( $k = 0 ; $k <= $#unique ; $k++ ) {
                $i = $k if ( $el eq $unique[$k] );
            }
            if ( $i == -1 ) {
                push ( @unique, $el );
                push ( @anzahl, 1 );
            }
            else { $anzahl[$i]++; }
        }
    }

    # Variablen kopieren
    foreach (@unique) { push ( @$werteref,  $_ ); }
    foreach (@anzahl) { push ( @$anzahlref, $_ ); }

    return ( $werteref, $anzahlref );
}

# ------------------------------------------------
# ---           build_index_file_name          ---
# ------------------------------------------------
# --- aus query Name des Indexfiles erzeugen ---
# ------------------------------------------------

sub build_index_file_name {

    # @_: bedingungen

    my @bed             = @_;
    my $index_file_name = "";
    foreach (@bed) { $index_file_name .= $_ . "."; }

    $index_file_name .= "idx";

    $index_file_name =~ s/\//__/g;

    # ersten _ löschen
    substr( $index_file_name, 0, 1 ) = "" if ( $index_file_name =~ /^_/ );
    substr( $index_file_name, 0, 1 ) = "" if ( $index_file_name =~ /^_/ );

    return $index_file_name;
}

# -------------------------------------
# ---          test_index_file        ---
# -------------------------------------
```

```
# --- Test, ob Index-File existiert ---
# -------------------------------------

sub test_index_file {

    # format des Indexfiles: ausgabe.bedingung.bedingung....idx
    # / werden durch _ ersetzt
    # [0] = xql-pfad
    # [1] = bedingung

    my $ausgabe = $_[0];
    my (@bed) = split ( /\s/, $_[1] );

    my @pseudo;

    my $bedref    = \@bed;
    my $pseudoref = \@pseudo;

    unshift ( @bed, $ausgabe );

    # unique starten, um doppelte namen im index zu vermeiden
    ( $bedref, $pseudoref ) = uniq( \@bed, \@pseudo );

    my $index_file_name = build_index_file_name(@bed);
    $index_file_name =~ s/\//_/g;

    substr( $ausgabe, 0, 1 ) = "" if ( $ausgabe =~ /^_/ );    # ersten _
                löschen

    while ( $index_file_name =~ /^_/ ) {
        substr( $index_file_name, 0, 1 ) = "";
    }

    my $index_checker = exist_index_file($index_file_name);

    my @werte;
    push ( @werte, $index_checker );
    push ( @werte, $index_file_name );

    return @werte;
}

# ------------------------
# --- exist_index_file ---
# ------------------------
# ---  Test ausführen  ---
```

```
# -------------------------

sub exist_index_file {

    # @_[0]: dateiname ohne verzeichnisprefix

    my $dateiname = $ENV{VAR_PATH} . $_[0];
    print "\nExistiert $dateiname???";
    if ( -e $dateiname ) {
        print "\nIndexfile existiert!";

        # -> Suche mit Sprite
        return (1);
    }
    else {
        print "\nIndexfile ex. nicht!";

        # -> Indexfile anlegen
        return (0);
    }
}

# --------------------------
# ---     logfile        ---
# --------------------------
# ---- Logfile schreiben ---
# --------------------------

sub logging {

    # %hash

    my %hash = @_;

    # Test, ob Logfile schon existiert
    unless ( -e $ENV{LOG} ) {
        open( LOGFILE, ">$ENV{LOG}" );
        print STDOUT "\nLogdatei existiert nicht -> neu anlegen";
        print LOGFILE "Pfade\tZeitpunkt\tDauer\n";
        close(LOGFILE);
    }

    my (
        $Sekunden, $Minuten,   $Stunden,   $Monatstag, $Monat,
        $Jahr,     $Wochentag, $Jahrestag, $Sommerzeit
      )
```

```
      = localtime(time);
    $Jahrestag += 1;

    open( LOGFILE, ">>$ENV{LOG}" );
    print LOGFILE "$hash{paths}\t$Jahrestag\t$hash{searchtime}\n";

    close(LOGFILE);
}

# ---------------------------------------------------
# ---                   average                   ---
# ---------------------------------------------------
# --- Mittelwert der Werte eines Feldes berechnen ---
# ---------------------------------------------------

sub average {

    # @_: feld der wertw

    my @things = @_;
    my $sum    = 0;
    foreach (@things) { $sum += $_; }
    return ( $sum / scalar @things );    #/
}

# ---------------------------------------------
# ---              build_query                ---
# ---------------------------------------------
# --- baut Query für die xql-Suche ...[...] ---
# ---------------------------------------------

sub build_query {
    my @path  = @_;
    my $query = "";

    return "__" . "$path[0]" . "[__" . "$path[0]" . "]"
      if ( scalar @path == 1 );

    for (@path) { print "\n506-path" . $_; }

    for ( my $el = 0 ; $el < scalar @path ; $el++ ) {
        unless ( $path[$el] =~ /^_/ ) {
            substr( $path[$el], 0, 0 ) = "__";
        }    # ersten _ einfügen

        $query .= "␣\$and\$␣" if ( $el > 1 );
```

```
        $query .= $path[$el];

        $query .= "[" if ( $el == 0 );
    }

    $query .= "]";
    return $query;
}

# -----------------------------------------
# ---          indexsuche                 ---
# -----------------------------------------
# --- Baut Query für xql-Suche --- --- --- ---
# --- Schreibt Ergebnisse in das Indexfile ---
# -----------------------------------------

sub indexsuche {

    # $_[0]: $index_file_name
    # $_[1]: $xml-file(s)

    my $index_file_name = $_[0];
    my $FILES           = $_[1];

    my (@splitters) = split ( /\./, $index_file_name );    #/
    pop (@splitters);

    # --- in splitters ist je Element ein Pfad
    # --- z.B. /bookstore/book/author/last_name /bookstore/index
    # --- XQL-Suche /bookstore/book/author/last_name[/bookstore/index]
    # --- und /bookstore/iIDndex[/bookstore/book/author/last_name]...

    # --- Query für erste Suche vorbereiten

    my @query;
    my @search;
    my @ilesen;
    my @index;
    my $rounds = 0;

    for ( $rounds = 0 ; $rounds < scalar @splitters ; $rounds++ ) {

        $query[$rounds]  = build_query(@splitters);
        $search[$rounds] = $splitters[0];
```

```
        print "\nAufruf von xql:: ";
        print "\nxql($query[$rounds],$search[$rounds],$ENV{XML_PATH})";

        my %hash;
        $hash{programm} = "xql.pl";
        $hash{xqlquery} = "\"" . $query[$rounds] . "\"";
        $hash{ausgabe}  = $search[$rounds];

        $hash{database} = $ENV{XML_PATH};
        $hash{xqlquery} =~ s/__/\//g;
        $hash{ausgabe}  =~ s/__/\//g;

        startProgram(%hash);

        # ----- ----- ----- ----- ----- ----- -----
        #      xql($query[$rounds],$search[$rounds],$FILES);
        # ----- ----- ----- ----- ----- ----- -----

        # ERG = file \t wert \t pfad
        # --- Ergebnis der ersten Suche speichern
        open( FIRST, "$ENV{VAR_PATH}xml.txt" );
        $ilesen[$rounds] = <FIRST>;
        close(FIRST);

        # ----------------------------------------
        # --- Query für nächste Suche vorbereiten ---

        # --- Elemente von Splitters umstellen
        my $last2first = shift (@splitters);

        push ( @splitters, $last2first );

        my $test = `cp $ENV{VAR_PATH}xml.txt $ENV{VAR_PATH}xml-$rounds.txt
                    `;

    }
    return prepare_index( $rounds, $index_file_name, $FILES );
}

# ---------------------
# --- prepare_index ---
# ---------------------

sub prepare_index {

    # $_[0] = $rounds
```

```
    # $_[1] = $index_file_name
    # $_[2] = xml-Pfad oder File

    # !!! no strict !!!
    no strict;

    # -----------------

    # i = aktuelle Datei
    # j = aktuelle Zeile

    # @XML_File;    # Feld, in dem die Dateinamen stehen

    my $anzahl_files  = $_[0];
    my $name_of_index = $_[1];
    my $XML_File      = $_[2];
    my @result;

    # dynamische Erstellung der XML-i-Variablen
    # XML-i enthalten alle Zeilen der Ergebnisse der letzten xql-Suche

    for ( my $i = 0 ; $i < $anzahl_files ; $i++ ) {
        open( XMLFILE, "$ENV{VAR_PATH}xml-$i.txt" );
        my $eval = "\@XML_$i = <XMLFILE>; shift \@XML_$i;";
        eval $eval;
        close(XMLFILE);
    }

    # alle XML-i zeilenweise aufsplitten
    $i = 0;
    for ( $i = 0 ; $i < $anzahl_files ; $i++ ) {

        # Anzahl der Zeilen der aktuellen Datei bestimmen
        my $eval = "\$M = scalar \@XML_$i;";
        eval $eval;

        #      $M = scalar @XML_0;

        for ( my $j = 0 ; $j < $M ; $j++ ) {

            # akt. Zeile nach \t aufsplitten
            my $eval = "\$text = \$XML";
            $eval .= "_";
            $eval .= "$i";
            $eval .= "[\$j]";
            eval $eval;
```

```
365             my (@test) = split ( /\t/, $text );

                if ( $i == 0 )     # erste Datei
                {

370                 # Dateinamen in XML-File schreiben
                    push ( @XML_File, $test[0] );
                }

                $eval = "push(";
375             $eval .= "\@XML";
                $eval .= "_";
                $eval .= "$i";
                $eval .= "_";
                $eval .= "Wert,$test[1]);";
380             eval $eval;
            }
        }

        # IndexFile schreiben
385
        # -------------- FilePrint ---------------------
        # --- bei *.xml wird der ganze Index neu erzeugt
    # --- bei einer einzelnen Datei wird nur das Ergebnis an den Index
                angehängt!

390     my $open = "";

        if ( $XML_File =~ /\*/ ) {

            # erzeugen einer neuen indexdatei
395
            $open = ">$ENV{VAR_PATH}$name_of_index";

            open( IDX, "$open" );
            select IDX;
400
            if ( $XML_File =~ /\*/ ) {

                # print Header
                my (@test) = split ( /\./, $name_of_index );    #/
405
                pop (@test);

                print "Filename";
```

```
            for (@test) {
                $_ =~ s/^__//g;
                print "\t" . $_;
            }
            print "\n";
        }

        # print Daten
        for ( my $j = 0 ; $j < $M ; $j++ ) {
            print $XML_File[$j];

            my $eval = "";
            my $text = "";

            for ( my $i = 0 ; $i < $anzahl_files ; $i++ ) {
                $eval = "\$XML";
                $eval .= "_";
                $eval .= "$i";
                $eval .= "_";
                $eval .= "Wert[$j];";

                $text .= "\t";
                $text .= eval $eval;
            }

            print $text;
            print "\n";
        }

        close(IDX);

        # -------------- FilePrint ---------------------
        select STDOUT;
    }
    else {

        # update einer datei
        # print Daten

        for ( my $j = 0 ; $j < $M ; $j++ ) {
            my $text = $XML_File[$j];

            my $eval = "";

            for ( my $i = 0 ; $i < $anzahl_files ; $i++ ) {
```

```
                $text .= "\t";
                $eval = "\$XML";
                $eval .= "_";
                $eval .= "$i";
                $eval .= "_";
                $eval .= "Wert[$j];";

                $text .= eval $eval;
            }

            push ( @result, $text . "\n" );
        }

    }

    for ( $i = 0 ; $i < $anzahl_files ; $i++ ) {
        $z = "rm␣$ENV{VAR_PATH}xml\-$i.txt";
        `$z`;
    }
    return @result;
}

sub pm {
    print "\n---␣---";
    my $feld = shift (@_);
    for (@_) { print "\n$feld:␣$_"; }
}

1;
```

C.5 Die XML-Testfiles

Mit XML-Files der Form wie sie in den folgenden Listings dargestellt sind, wurden die verschiedenen Testreihen aus Kapitel 6 auf Seite 83 durchgeführt. Die Verschachtelungstiefe ist immer ohne das `root`-Element anzusehen, da dieses in allen XML-Dokumenten enthalten ist.

Listing C.13: XML-File mit Verschachtelungstiefe 1

```
<?xml version='1.0'?>
<root>
     <element1>1000</element1>
     <element2>2000</element2>
     <element3>3000</element3>
     <element4>4000</element4>
     <element5>5000</element5>
     <element6>6000</element6>
     <element7>7000</element7>
     <element8>8000</element8>
     <element9>9000</element9>
     <elementA>AAAA</elementA>
     <elementB>BBBB</elementB>
     <elementC>CCCC</elementC>
     <elementD>DDDD</elementD>
     <elementE>EEEE</elementE>
     <elementF>FFFF</elementF>
</root>
```

Listing C.14: XML-File mit Verschachtelungstiefe 2

```
<?xml version='1.0'?>
<root>
     <element1>
          <element11>1000</element11>
          <element12>2000</element12>
     </element1>
     <element2>2000</element2>
     <element3>
          <element31>3000</element31>
          <element32>4000</element32>
     </element3>
     <element4>4000</element4>
     <element5>
          <element51>5000</element51>
          <element52>AAAA</element52>
          <element53>BBBB</element53>
          <element54>CCCC</element54>
     </element5>
```

```
19      <element6>6000</element6>
20      <element7>7000</element7>
21      <element8>8000</element8>
22      <element9>9000</element9>
23      <elementA>AAAA</elementA>
24      <elementB>BBBB</elementB>
25      <elementC>CCCC</elementC>
26      <elementD>DDDD</elementD>
27      <elementE>EEEE</elementE>
28      <elementF>FFFF</elementF>
29  </root>
```

Listing C.15: XML-File mit Verschachtelungstiefe 3

```
1   <?xml version='1.0'?>
2   <root>
3       <element1>
4           <element11>
5               <element111>4000</element111>
6               <element111>5000</element111>
7           </element11>
8           <element12>2000</element12>
9       </element1>
10      <element2>2000</element2>
11      <element3>
12          <element31>
13              <element311>7000</element311>
14              <element311>8000</element311>
15          </element31>
16          <element32>4000</element32>
17      </element3>
18      <element4>4000</element4>
19      <element5>
20          <element51>
21              <element511>AAAA</element511>
22              <element511>BBBB</element511>
23          </element51>
24          <element52>AAAA</element52>
25          <element53>BBBB</element53>
26          <element54>CCCC</element54>
27      </element5>
28      <element6>6000</element6>
29      <element7>7000</element7>
30      <element8>8000</element8>
31      <element9>9000</element9>
32      <elementA>AAAA</elementA>
33      <elementB>BBBB</elementB>
```

```
34      <elementC>CCCC</elementC>
35      <elementD>DDDD</elementD>
36      <elementE>EEEE</elementE>
37      <elementF>FFFF</elementF>
38  </root>
```

Listing C.16: XML-File mit Verschachtelungstiefe 4

```
1   <?xml version='1.0'?>
2   <root>
3       <element1>
4           <element11>
5               <element111>
6                   <element1114>3000</element1114>
7                   <element1115>6000</element1115>
8               </element111>
9               <element111>5000</element111>
10          </element11>
11          <element12>2000</element12>
12      </element1>
13      <element2>2000</element2>
14      <element3>
15          <element31>
16              <element311>
17                  <element3114>9000</element3114>
18              </element311>
19              <element311>
20                  <element3115>CCCC</element3115>
21              </element311>
22          </element31>
23          <element32>4000</element32>
24      </element3>
25      <element4>4000</element4>
26      <element5>
27          <element51>
28              <element511>AAAA</element511>
29              <element511>BBBB</element511>
30          </element51>
31          <element52>AAAA</element52>
32          <element53>BBBB</element53>
33          <element54>CCCC</element54>
34      </element5>
35      <element6>6000</element6>
36      <element7>7000</element7>
37      <element8>8000</element8>
38      <element9>9000</element9>
39      <elementA>AAAA</elementA>
```

```
40      <elementB>BBBB</elementB>
41      <elementC>CCCC</elementC>
42      <elementD>DDDD</elementD>
43      <elementE>EEEE</elementE>
44      <elementF>FFFF</elementF>
45  </root>
```

Erklärung

"Ich versichere, dass ich die vorliegende Arbeit selbstständig und nur unter Verwendung der angegebenen Quellen und Hilfsmittel angefertigt habe."

Leipzig, den

Colin Hotzky